



BEATuxedo®

BEA Tuxedo CORBA サーバ・アプリケーションの開発方法

BEA Tuxedo 8.1
2003 年 1 月

Copyright

Copyright © 2003 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Liquid Data for WebLogic, BEA Manager, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop and How Business Becomes E-Business are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

目次

このマニュアルについて

対象読者.....	xii
e-docs Web サイト.....	xiii
マニュアルの印刷方法.....	xiii
関連情報.....	xiii
サポート情報.....	xiv
表記上の規則.....	xv

1. CORBA サーバ・アプリケーションの概念

CORBA サーバ・アプリケーションをビルドするために作成するエンティティ.....	1-2
サーバ・アプリケーション用の CORBA オブジェクトのインプリメンテーション.....	1-2
OMG IDL インターフェイス定義で CORBA オブジェクトのオペレーションを確立する方法.....	1-3
CORBA オブジェクトのオペレーションをインプリメントする方法.....	1-4
クライアント・アプリケーションがサーバ・アプリケーションの CORBA オブジェクトをアクセスおよび操作する方法.....	1-5
CORBA オブジェクトの実行時のインスタンス化.....	1-8
Server オブジェクト.....	1-10
CORBA サーバ・アプリケーションの開発プロセス.....	1-11
オブジェクト・リファレンスの生成.....	1-11
クライアント・アプリケーションがサーバ・アプリケーションのファクトリを見つけるしくみ.....	1-12
アクティブ・オブジェクト・リファレンスの作成.....	1-12
オブジェクト状態の管理.....	1-13
オブジェクト状態について.....	1-14
オブジェクトの活性化方針.....	1-15
アプリケーション制御の非活性化.....	1-17
オブジェクトのデータの読み取りと書き込み.....	1-19
オブジェクトの永続状態を読み書きするためのメカニズム.....	1-20

オブジェクトの活性化時の状態の読み取り	1-25
オブジェクトの個々のオペレーションでの状態の読み取り	1-26
状態を持たないオブジェクトと永続状態	1-26
状態を持つオブジェクトと永続状態	1-27
オブジェクトの非活性化の責任	1-28
不必要な I/O の回避	1-29
活性化のサンプル	1-29
デザイン・パターンの使い方	1-30
Process-Entity デザイン・パターン	1-30
List-Enumerator デザイン・パターン	1-31

2. BEA Tuxedo CORBA サーバ・アプリケーションの作成手順

CORBA サーバ・アプリケーション開発プロセスの概要	2-2
ステップ 1: サーバ・アプリケーション用の OMG IDL ファイルのコンパイル	
2-3	
IDL コンパイラの使い方	2-5
スケルトンとインプリメンテーション・ファイルの生成	2-7
tie クラスの生成	2-7
ステップ 2: 各インターフェイスのオペレーションをインプリメントするメソッドの記述	2-8
IDL コンパイラによって生成されるインプリメンテーション・ファイル	
2-9	
ファクトリのインプリメント	2-9
ステップ 3: サーバ・オブジェクトの作成	2-11
サーバ・アプリケーションの初期化	2-12
ファクトリを作成および登録するコードの記述	2-13
サーバントの作成	2-15
サーバ・アプリケーションのリリース	2-16
ステップ 4: オブジェクトのメモリ内での振る舞い	2-18
ICF でオブジェクトの活性化方針とトランザクション方針を指定する方法	2-19
ステップ 5: サーバ・アプリケーションのコンパイルとリンク	2-22
ステップ 6: サーバ・アプリケーションのデプロイ	2-23
開発とデバッグのヒント	2-24
CORBA 例外とユーザ・ログの使い方	2-25

例外のクライアント・アプリケーション・ビュー	2-25
例外のサーバ・アプリケーション・ビュー	2-25
Detecting Error Conditions in the Callback Methods	2-31
OMG IDL インターフェイスのバージョンと修正に関する一般的な注意 事項	2-32
Tobj_ServantBase::deactivate_object() での状態処理に関する注意	2-33
サーバント・プール	2-34
サーバント・プールの機能	2-35
サーバント・プールのインプリメント方法	2-35
デレゲーション・ベースのインターフェイス・インプリメンテーション	2-36
BEA Tuxedo システムの tie クラスについて	2-37
tie クラスを使用する条件	2-39
CORBA アプリケーションに tie クラスを作成する方法	2-40

3. Basic CORBA サーバ・アプリケーションの設計とインプリメント

Basic University サンプル・アプリケーションのしくみ	3-2
Basic University サンプル・アプリケーションの OMG IDL	3-3
アプリケーションの起動	3-5
コース概要の参照	3-5
コース詳細の参照	3-8
University サーバ・アプリケーションの設計上の考慮事項	3-9
オブジェクト・リファレンスの生成に関する設計上の考慮事項	3-9
オブジェクト状態の管理に関する設計上の考慮事項	3-12
RegistrarFactory オブジェクト	3-12
Registrar オブジェクト	3-12
CourseSynopsisEnumerator オブジェクト	3-13
Basic University サンプル・アプリケーションの ICF ファイル	3-14
永続状態情報の処理に関する設計上の考慮事項	3-14
Registrar オブジェクト	3-15
CourseSynopsisEnumerator オブジェクト	3-15
University データベースの使い方	3-16
Basic サンプル・アプリケーションがデザイン・パターンを適用する方 法	3-17
Process-Entity デザイン・パターン	3-18
List-Enumerator デザイン・パターン	3-18

BEA Tuxedo システムに組み込まれた性能効率化.....	3-20
状態を持つオブジェクトの事前活性化	3-20
状態を持つオブジェクトを事前活性化する方法.....	3-21
事前活性化オブジェクトの使い方に関する注意.....	3-21

4. マルチスレッド CORBA サーバ・アプリケーションの作成

概要.....	4-24
はじめに.....	4-24
要件、目標、概念	4-26
スレッド・モデル.....	4-28
リエントラント・サーバント	4-30
Current オブジェクト.....	4-31
マルチスレッド CORBA サーバをサポートするためのメカニズム.....	4-32
コンテキスト・サービス.....	4-32
TP フレームワークのクラスとメソッド	4-33
build コマンドの機能.....	4-34
管理用ツール.....	4-35
マルチスレッド・システムでのシングル・スレッド・サーバ・アプリケーションの実行	4-35
マルチスレッド CORBA サーバ・アプリケーションの開発とビルド	4-36
buildobjserver コマンドの使い方	4-36
プラットフォーム固有のスレッド・ライブラリ.....	4-36
マルチスレッド・サポートの指定	4-37
代替サーバ・クラスの指定.....	4-37
buildobjclient コマンドの使い方	4-38
非リエントラント・サーバントの作成	4-39
リエントラント・サーバントの作成.....	4-41
クライアント・アプリケーションに関する考慮事項	4-41
マルチスレッド simpapp サンプル・アプリケーションのビルドと実行...	4-43
simpapp マルチスレッド・サンプルについて	4-43
サンプル・アプリケーションの機能.....	4-43
simpapp マルチスレッド・サンプル・アプリケーションの OMG IDL コード	4-45
サンプル・アプリケーションのビルドと実行の方法	4-46
TUXDIR 環境変数の設定.....	4-47

TUXDIR 環境変数の検証.....	4-48
環境変数の設定の変更	4-48
サンプル・アプリケーションの作業ディレクトリの作成	4-49
すべてのファイルのパーミッションのチェック	4-53
Executing the runme Command	4-54
サンプル・アプリケーションのステップ・バイ・ステップ実行.....	4-59
サンプル・アプリケーションのシャットダウン.....	4-62
マルチスレッド CORBA サーバ・アプリケーションの管理.....	4-63
スレッド・プール・サイズの指定	4-63
MAXDISPATCHTHREADS	4-63
MINDISPATCHTHREADS.....	4-65
スレッド・モデルの指定.....	4-65
活性化されたオブジェクトの数の指定.....	4-66
UBBCONFIG ファイルの例.....	4-67

5. セキュリティと CORBA サーバ・アプリケーション

セキュリティと CORBA サーバ・アプリケーションの概要	5-2
University サーバ・アプリケーションの設計上の考慮事項.....	5-3
Security University サンプル・アプリケーションのしくみ.....	5-4
クライアント・アプリケーションに学生の詳細情報を返す機能を設計する際の考慮事項.....	5-6

6. トランザクションの CORBA サーバ・アプリケーションへの統合

BEA Tuxedo システムでのトランザクションの概要.....	6-2
CORBA サーバ・アプリケーションでのトランザクションの設計およびインプリメンテーション	6-5
Transactions University サンプル・アプリケーションのしくみ.....	6-7
Transactions University サンプル・アプリケーションで使用するトランザクション・モデル.....	6-8
University サーバ・アプリケーションのオブジェクト状態に関する注意事項	6-9
Registrar オブジェクトで定義されたオブジェクト方針.....	6-10
RegistrarFactory オブジェクトで定義されたオブジェクト方針 ...	6-10
Transactions サンプル・アプリケーションでの XA リソース・マネージャの使用.....	6-11

Transactions サンプル・アプリケーションのコンフィギュレーションの要件	6-11
トランザクションの CORBA クライアントおよびサーバ・アプリケーションへの統合	6-12
オブジェクトを自動的にトランザクションに関与させる方法	6-14
オブジェクトのトランザクションへの参加の有効化	6-15
トランザクションのスコープ指定時のオブジェクト呼び出しの防止	6-16
実行中のトランザクションからのオブジェクトの除外	6-17
方針の割り当て	6-18
XA リソース・マネージャのオープン	6-18
XA リソース・マネージャのクローズ	6-18
トランザクション管理とオブジェクト状態管理	6-19
XA リソース・マネージャへのオブジェクト状態管理の委譲	6-19
トランザクションの作業が完了してから、データベースへの書き込みが始まるまでの待機	6-20
BEA Tuxedo のトランザクションの使用に関する注意事項	6-22
ユーザ定義の例外	6-24
例外の定義	6-25
例外のスロー	6-25

7. BEA Tuxedo サービスの CORBA オブジェクトへのラッピング

BEA Tuxedo サービスのラッピングの概要	7-2
BEA Tuxedo サービスをラッピングするオブジェクトの設計	7-3
BEA Tuxedo サービス呼び出しをカプセル化するバッファの作成	7-5
BEA Tuxedo サービスとの間でメッセージを送信するオペレーションのインプリメンテーション	7-6
制限事項	7-8
Wrapper サンプル・アプリケーションの設計上の考慮事項	7-9
Wrapper University サンプル・アプリケーションのしくみ	7-11
Billing サーバ・アプリケーションのインターフェイス定義	7-12
Wrapper サンプル・アプリケーションの設計上の追加考慮事項	7-12

8. BEA Tuxedo の CORBA サーバ・アプリケーションのスケールリング

BEA Tuxedo システムで利用可能なスケラビリティ機能の概要	8-2
---	-----

BEA Tuxedo サーバ・アプリケーションのスケールリング	8-3
Production サンプル・アプリケーションの OMG IDL の変更	8-5
サーバ・プロセスおよびサーバ・グループの複製	8-5
複製されたサーバ・プロセス	8-6
複製されたサーバ・グループ	8-8
複製されたサーバ・プロセスおよびグループのコンフィギュレーション	8-9
オブジェクト状態管理によるアプリケーションのスケールリング	8-11
ファクトリ・ベース・ルーティング	8-14
ファクトリ・ベース・ルーティングのしくみ	8-15
UBBCONFIG ファイルでのファクトリ・ベース・ルーティングのコンフィギュレーション	8-16
ファクトリでのファクトリ・ベース・ルーティングのインプリメンテーション	8-19
実行時の処理	8-21
Registrar および Teller オブジェクトに関する設計上の追加考慮事項	8-22
Registrar および Teller オブジェクトのインスタンス化	8-22
適切なサーバ・グループで学生の登録が発生するようにする方法	8-24
Teller オブジェクトが適切なサーバ・グループでインスタンス化されるようにする方法	8-26
Production サーバ・アプリケーションをさらにスケールリングする方法	8-27
状態を持たないオブジェクトまたは状態を持つオブジェクトの選択	8-28
状態を持たないオブジェクトが必要な場合	8-29
状態を持つオブジェクトが必要な場合	8-30

索引



このマニュアルについて

このマニュアルでは、プログラマが BEA Tuxedo 製品の主な機能をインプリメントして、BEA Tuxedo ドメインで実行されるスケラブルで高性能なサーバ・アプリケーションを設計およびインプリメントする方法について説明しています。「第 3 章 Basic CORBA サーバ・アプリケーションの設計とインプリメント」で使用されている例は、『BEA Tuxedo CORBA University サンプル・アプリケーション』で説明するサンプル・アプリケーションに基づいています。

このマニュアルでは、以下の内容について説明します。

- 「第 1 章 CORBA サーバ・アプリケーションの概念」では、BEA Tuxedo サーバ・アプリケーションの作成に関する基本的な概念を示し、BEA Tuxedo サーバ・アプリケーションをビルドするために作成する 2 つの主要なプログラミング・エンティティについて説明します。
- 「第 2 章 BEA Tuxedo CORBA サーバ・アプリケーションの作成手順」では、BEA Tuxedo サーバ・アプリケーションを作成する基本的な手順について説明します。
- 「第 3 章 Basic CORBA サーバ・アプリケーションの設計とインプリメント」では、CORBA サーバ・アプリケーションの設計およびインプリメントに関連する基本的な概念と手順について、Basic University サンプル・アプリケーションに基づいて説明します。
- 「第 4 章 マルチスレッド CORBA サーバ・アプリケーションの作成」では、マルチスレッド CORBA サーバ・アプリケーションを作成するための要件、目的、および概念について概説します。また、この章では、マルチスレッド CORBA サーバ・アプリケーションを開発およびビルドする手順について概説します。また、simpapp_mt サンプル・アプリケーションをビルドおよび実行する方法と、マルチスレッド CORBA サーバ・アプリケーションを管理する方法についても説明します。

-
- 「第 5 章 セキュリティと CORBA サーバ・アプリケーション」では、BEA Tuxedo ドメインにセキュリティ・モデルをインプリメントする際に、CORBA サーバ・アプリケーションが果たす役割について説明します。
 - 「第 6 章 トランザクションの CORBA サーバ・アプリケーションへの統合」では、BEA Tuxedo システムによってサポートされている BEA Tuxedo ドメインのトランザクション、およびサーバ・アプリケーションにトランザクションをインプリメントする方法について説明します。
 - 「第 7 章 BEA Tuxedo サービスの CORBA オブジェクトへのラッピング」では、BEA Tuxedo アプリケーションを CORBA サーバ・アプリケーションに統合する方法について説明します。
 - 「第 8 章 BEA Tuxedo の CORBA サーバ・アプリケーションのスケールアップ」では、複製されたサーバ・プロセスとサーバ・グループ、ファクトリ・ベース・ルーティング、およびオブジェクトの状態管理を含む主要なスケラビリティ機能を BEA Tuxedo アプリケーションに組み込んで、スケラビリティを向上させる方法について説明します。

対象読者

このマニュアルは、安全でスケラブルなトランザクション・ベースのサーバ・アプリケーションを作成するプログラムを対象としています。BEA Tuxedo システム、CORBA、および C++ プログラミングに読者が精通していることを前提として書かれています。

e-docs Web サイト

BEA Tuxedo 製品のマニュアルは BEA 社の Web サイトで参照することができます。BEA ホーム・ページの [製品のドキュメント] をクリックするか、または <http://edocs.beasys.co.jp/e-docs/index.html> に直接アクセスしてください。

マニュアルの印刷方法

このマニュアルは、ご使用の Web ブラウザで一度に 1 ファイルずつ印刷できます。Web ブラウザの [ファイル] メニューにある [印刷] オプションを使用してください。

このマニュアルの PDF 版は、Web サイト上にあります。また、マニュアルの CD-ROM にも収められています。BEA Tuxedo この PDF を Adobe Acrobat Reader で開くと、マニュアル全体または一部をブック形式で印刷できます。PDF 形式を利用するには、BEA Tuxedo マニュアルのホーム・ページにある [PDF 版] ボタンをクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader をお持ちでない場合は、Adobe Web サイト (<http://www.adobe.co.jp/>) から無償でダウンロードできます。

関連情報

CORBA、BEA Tuxedo、分散オブジェクト・コンピューティング、トランザクション処理、C++ プログラミング、および Java プログラミングの詳細については、BEA Tuxedo オンライン・マニュアルの「Bibliography」を参照してください。

サポート情報

皆様の BEA Tuxedo マニュアルに対するフィードバックをお待ちしています。ご意見やご質問がありましたら、電子メールで docsupport-jp@bea.com までお送りください。お寄せいただきましたご意見は、BEA Tuxedo マニュアルの作成および改訂を担当する BEA 社のスタッフが直接検討いたします。

電子メール メッセージには、BEA Tuxedo 8.0 リリースのマニュアルを使用していることを明記してください。

BEA Tuxedo に関するご質問、または BEA Tuxedo のインストールや使用に際して問題が発生した場合は、www.bea.com の BEA WebSUPPORT を通して BEA カスタマ・サポートにお問い合わせください。カスタマ・サポートへの問い合わせ方法は、製品パッケージに同梱されているカスタマ・サポート・カードにも記載されています。

カスタマ・サポートへお問い合わせの際には、以下の情報をご用意ください。

- お客様のお名前、電子メール・アドレス、電話番号、Fax 番号
- お客様の会社名と会社の住所
- ご使用のマシンの機種と認証コード
- ご使用の製品名とバージョン
- 問題の説明と関連するエラー・メッセージの内容

表記上の規則

このマニュアルでは、以下の表記規則が使用されています。

規則	項目
太字	用語集に定義されている用語を示します。
Ctrl + Tab	2 つ以上のキーを同時に押す操作を示します。
イタリック体	強調またはマニュアルのタイトルを示します。
等幅テキスト	コード・サンプル、コマンドとオプション、データ構造とメンバ、データ型、ディレクトリ、およびファイル名と拡張子を示します。また、キーボードから入力するテキストも等幅テキストで表示します。 例： <pre>#include <iostream.h> void main () the pointer psz chmod u+w * \tux\data\ap .doc tux.doc BITMAP float</pre>
太字の等幅テキスト	コード内の重要な語を示します。 例： <pre>void commit ()</pre>
斜体の等幅テキスト	コード内の変数を示します。 例： <pre>String <i>expr</i></pre>

規則	項目
大文字のテキスト	<p>デバイス名、環境変数、および論理演算子を示します。</p> <p>例：</p> <p>LPT1</p> <p>SIGNON</p> <p>OR</p>
{ }	<p>構文の行で、選択肢の組み合わせを示します。かっこは入力しません。</p>
[]	<p>構文の行で、オプション項目を示します。かっこは入力しません。</p> <p>例：</p> <pre>buildobjclient [-v] [-o name] [-f file-list]... [-l file-list]...</pre>
	<p>構文の行で、相互に排他的な選択肢の区切りとして使います。記号は入力しません。</p>
...	<p>コマンド・ラインで、以下のいずれかの場合を示します。</p> <ul style="list-style-type: none"> ■ コマンド・ラインで、同じ引数を繰り返し使用できることを示します。 ■ 文で、追加のオプション引数が省略されていることを示します。 ■ 追加のパラメータ、値、またはその他の情報を入力できることを示します。 <p>記号は入力しません。</p> <p>例：</p> <pre>buildobjclient [-v] [-o name] [-f file-list]... [-l file-list]...</pre>
.	<p>コード例または構文の行で、項目が省略されていることを示します。記号は入力しません。</p>

1 CORBA サーバ・アプリケーションの概念

ここでは、以下の内容について説明します。

- CORBA サーバ・アプリケーションをビルドするために作成するエンティティ：
 - サーバ・アプリケーション用の CORBA オブジェクトのインプリメンテーション
 - Server オブジェクト
- CORBA サーバ・アプリケーションの開発プロセス：
 - オブジェクト・リファレンスの生成
 - オブジェクト状態の管理
 - オブジェクトのデータの読み取りと書き込み
 - デザイン・パターンの使い方

このマニュアルの各章では、さまざまな BEA Tuxedo ソフトウェア機能を活用する CORBA サーバ・アプリケーションをビルドする手順および例について説明します。BEA Tuxedo CORBA サーバ・アプリケーションとその機能に関する基本情報については、『BEA Tuxedo CORBA アプリケーション入門』を参照してください。

CORBA サーバ・アプリケーションをビルドするために作成するエンティティ

CORBA サーバ・アプリケーションをビルドするには、次の2つのエンティティを作成します。

- サーバ・アプリケーションのビジネス・ロジックを実行する CORBA オブジェクトのインプリメンテーション
- サーバ・アプリケーションを初期化およびリリースし、クライアント要求を満たすために必要な CORBA オブジェクトをインスタンス化するオペレーションをインプリメントする Server オブジェクト

また、IDL コンパイラによって生成され、CORBA サーバ・アプリケーションに組み込む複数のファイルも使用します。これらのファイルのリストと説明については、「第 2 章 BEA Tuxedo CORBA サーバ・アプリケーションの作成手順」を参照してください。

以降の節では、これらのエンティティの基礎知識について説明します。これらのエンティティの生成の詳細については、「第 2 章 BEA Tuxedo CORBA サーバ・アプリケーションの作成手順」を参照してください。

サーバ・アプリケーション用の CORBA オブジェクトのインプリメンテーション

CORBA オブジェクトとは何か、およびそれらをどのように定義、インプリメント、インスタンス化、管理するかについて明確に理解しておくことは、CORBA サーバ・アプリケーションの設計者および作成者にとって不可欠です。

Object Management Group インターフェイス定義言語 (OMG IDL) で定義したインターフェイスの CORBA オブジェクトには、CORBA サーバ・アプリケーションのビジネス・ロジックとデータが含まれます。すべてのクライアント・アプリケーション要求では、CORBA オブジェクトのオペレーション

が呼び出されます。インターフェイス用に定義されたオペレーションをインプリメントするコードを、オブジェクト・インプリメンテーションと言います。たとえば、C++ では、オブジェクト・インプリメンテーションは C++ クラスです。

ここでは、以下の内容について説明します。

- OMG IDL インターフェイス定義で CORBA オブジェクトの呼び出し可能なオペレーションを確立する方法
- CORBA オブジェクトのオペレーションをインプリメントする方法
- クライアント・アプリケーションがサーバ・アプリケーションの CORBA オブジェクトをアクセスおよび操作する方法
- クライアント要求への応答としてコードとデータを持つ CORBA オブジェクトを実行時にインスタンス化する方法

OMG IDL インターフェイス定義で CORBA オブジェクトのオペレーションを確立する方法

CORBA オブジェクトのインターフェイスは、そのオブジェクトで実行可能なオペレーションを識別します。CORBA オブジェクトに固有の特性は、オブジェクトのインターフェイス定義がそのインプリメンテーションから独立していることです。インターフェイス定義により、インターフェイスのオペレーションをインプリメントする方法が確立されます。これには、オペレーションに受け渡し、オペレーションから返される有効なパラメータなどが含まれます。

インターフェイス定義は OMG IDL で表現され、アプリケーションのクライアント / サーバ間の取り決めを確立します。つまり、特定のインターフェイスについて、サーバ・アプリケーションは次のことを行います。

- そのインターフェイスで定義されたオペレーションをインプリメントします。
- 常に各オペレーションで定義されたパラメータを使用します。

サーバ・アプリケーションがどのようにオペレーションをインプリメントするかは、時間と共に変わる場合があります。この振る舞いは、サーバ・アプリケーションが定義されたインターフェイスをインプリメントし、定義され

1 CORBA サーバ・アプリケーションの概念

たパラメータを使用するという要件を満たす限り許容されます。このため、クライアント・スタブは常にサーバ・マシン上のオブジェクト・インプリメンテーションの信頼できるプロキシであり続けます。これは、CORBA アーキテクチャの主要なメリットの1つです。つまり、サーバ・アプリケーションがオブジェクトをインプリメントする方法を変更するときに、クライアント・アプリケーションを修正する必要がなく、クライアント・アプリケーションがその変更を認識する必要もないということです。

また、インターフェイス定義は、クライアント・スタブと、サーバ・アプリケーションのスケルトンの両方の内容を決定します。これらの2つのエンティティにORBとポータブル・オブジェクト・アダプタ(POA)を組み合わせることにより、オブジェクトのオペレーションに対するクライアント要求を、その要求を満たすことができるサーバ・アプリケーションのコードにルーティングできます。

システム設計者がアプリケーションのビジネス・オブジェクトのインターフェイスを指定したら、プログラムはそのインターフェイスをインプリメントします。このマニュアルでは、その方法について説明します。

OMG IDLの詳細については、『BEA Tuxedo CORBA クライアント・アプリケーションの開発方法』を参照してください。

CORBA オブジェクトのオペレーションをインプリメントする方法

前述のとおり、インターフェイスで定義されたオペレーションをインプリメントするコードを、オブジェクト・インプリメンテーションと言います。C++の場合、このコードはメソッドのセットで構成され、アプリケーションのOMG IDLファイル内のインターフェイスで定義されたオペレーションごとに1つ存在します。アプリケーションのオブジェクト・インプリメンテーションのセットが記述されたファイルを、インプリメンテーション・ファイルと言います。BEA Tuxedo システムには、IDL コンパイラが用意されています。IDL コンパイラは、次の図に示すように、アプリケーションのOMG IDL ファイルをコンパイルして、インプリメンテーション・ファイルを含むいくつかのファイルを生成します。



生成されたインプリメンテーション・ファイルには、メソッド・テンプレート、メソッド宣言、オブジェクトのコンストラクタとデストラクタ、およびその他のデータが含まれています。これらを出発点として、アプリケーションのオブジェクト・インプリメンテーションを記述できます。たとえば C++ では、生成されたインプリメンテーション・ファイルには各インターフェイスのメソッドのシグニチャが含まれています。このファイルに各メソッドのビジネス・ロジックを入力し、そのファイルをサーバ・アプリケーションの実行可能ファイルをビルドするコマンドへの入力として提供します。

クライアント・アプリケーションがサーバ・アプリケーションの CORBA オブジェクトをアクセスおよび操作する方法

クライアント・アプリケーションは、サーバ・アプリケーションによって管理される CORBA オブジェクトのオブジェクト・リファレンスを介して、それらのオブジェクトをアクセスおよび操作します。クライアント・アプリケーションは、オブジェクト・リファレンス上のオペレーション（つまり要求）を呼び出します。これらの要求はメッセージとしてサーバ・アプリケーションに送信されます。サーバ・アプリケーションは、CORBA オブジェクトの適切なオペレーションを呼び出します。これらの要求がサーバ・アプリケーションに送信され、そのサーバ・アプリケーションで呼び出されるという事実は、クライアントからは完全に見えません。クライアント・アプリケーションは、単にクライアント・スタブで呼び出しを行っているように見えます。

クライアント・アプリケーションは、オブジェクト・リファレンスによってのみ CORBA オブジェクトを操作できます。設計上の主要な考慮事項の 1 つは、アプリケーションに適した方法でどのようにオブジェクト・リファレンスを作成し、それらを必要としているクライアント・アプリケーションにそれらを返すかです。

1 CORBA サーバ・アプリケーションの概念

一般に、CORBA オブジェクトへのオブジェクト・リファレンスはファクトリによって BEA Tuxedo システムで作成されます。ファクトリは、別の CORBA オブジェクトへのオブジェクト・リファレンスをそのオペレーションの 1 つとして返す任意の CORBA オブジェクトです。アプリケーションのファクトリは、アプリケーション用に定義したほかの CORBA オブジェクトと同じようにインプリメントします。ファクトリを FactoryFinder に登録することによって、ファクトリを BEA Tuxedo ドメイン、およびその BEA Tuxedo ドメインに接続されているクライアントに認識させることができます。ファクトリの登録は、通常 Server オブジェクトによって実行されるオペレーションです。詳細については、「第 1 章の 10 ページ「Server オブジェクト」」を参照してください。ファクトリの設計の詳細については、「第 1 章の 11 ページ「オブジェクト・リファレンスの生成」」を参照してください。

オブジェクト・リファレンスの内容

クライアント・アプリケーションからは、オブジェクト・リファレンスはオペーク、つまりブラック・ボックスのように見えます。クライアント・アプリケーションは、その内容を知らなくてもオブジェクト・リファレンスを使用できます。ただし、オブジェクト・リファレンスには、BEA Tuxedo システムが特定のオブジェクト・インスタンス、およびそのオブジェクトに関連付けられている状態データを検索するために必要なすべての情報が格納されています。

オブジェクト・リファレンスには、以下の情報が格納されます。

- インターフェイス名

これは、オブジェクトの OMG IDL インターフェイスのインターフェイス・リポジトリ ID です。

- オブジェクト ID (OID)

OID は、リファレンスを適用するオブジェクトのインスタンスを一意に識別します。オブジェクトが外部ストレージ内にデータを持っている場合、OID には通常サーバ・マシンがそのオブジェクトのデータを検索するために使用できるキーも含まれます。

- グループ ID

グループ ID は、クライアント・アプリケーションがオブジェクト・リファレンスを使用して要求を行ったときに、そのオブジェクト・リファ

レンスのルーティング先となるサーバ・グループを識別します。デフォルト以外のグループ ID の生成は、ファクトリ・ベース・ルーティングという BEA Tuxedo の主要機能の 1 つです。詳細については、「第 8 章の 14 ページ「ファクトリ・ベース・ルーティング」」で説明します。

注記 上のリストの 3 つの情報を組み合わせることにより、CORBA オブジェクトが一意に識別されます。指定されたインターフェイスおよび OID を持つ 1 つのオブジェクトが 2 つの異なるグループで同時に活性化されることも、それら 2 つのグループに同じオブジェクト・インプリメンテーションがある場合には起こりえます。指定されたインターフェイス名および OID の 1 つのオブジェクトがドメイン内で一度に 1 つだけ利用可能であることを保証する必要がある場合は、次のいずれかを行います。ファクトリ・ベース・ルーティングを使用して、特定の OID のオブジェクトが常に同じグループにルーティングされるようにするか、指定されるオブジェクト・インプリメンテーションが 1 つのグループのみに存在するようにドメインをコンフィグレーションする。そうすれば、指定されたインターフェイス名および OID を持つ 1 つのオブジェクトに複数のクライアントがオブジェクト・リファレンスをした場合でも、そのリファレンスは常に同じオブジェクト・インスタンスへルーティングされることになります。

ファクトリ・ベース・ルーティングの詳細については、「第 8 章の 14 ページ「ファクトリ・ベース・ルーティング」」を参照してください。

オブジェクト・リファレンスの存続期間

BEA Tuxedo ドメインで動作するサーバ・アプリケーションによって作成されるオブジェクト・リファレンスには、通常それを作成したサーバ・プロセスの存続期間より長い寿命が設定されます。BEA Tuxedo オブジェクト・リファレンスは、最初にそれらを作成したサーバ・プロセスが実行されているかどうかに関係なく、クライアント・アプリケーションで使用できます。このように、オブジェクト・リファレンスは特定のサーバ・プロセスに関連付けられてはいません。

1 CORBA サーバ・アプリケーションの概念

`TP::create_active_object_reference()` オペレーションで作成されたオブジェクト・リファレンスは、それを作成したサーバ・プロセスが存続する間だけ有効です。詳細については、第 3 章の 20 ページ「状態を持つオブジェクトの事前活性化」を参照してください。

オブジェクト・インスタンスの受け渡し

ORB は、オブジェクト・インスタンスをオブジェクト・リファレンスとしてマーシャリングできません。たとえば、次のコードでファクトリ・リファレンスを受け渡すと、BEA Tuxedo システムで CORBA マーシャリング例外が発生します。

```
connection::setFactory(this);
```

オブジェクト・インスタンスを受け渡すには、次の例のように、プロキシ・オブジェクト・リファレンスを作成して、そのプロキシを代わりに受け渡します。

```
CORBA::Object myRef = TP::get_object_reference();
ResultSetFactory factoryRef = ResultSetFactoryHelper::_narrow(myRef);
connection::setFactoryRef(factoryRef);
```

CORBA オブジェクトの実行時のインスタンス化

サーバ・アプリケーションが、サーバ・マシンのメモリにマップされていないオブジェクト（活性化されていないオブジェクト）の要求を受信した場合、TP フレームワークは `Server::create_servant()` オペレーションを呼び出します。`Server::create_servant()` オペレーションは、作成する CORBA サーバ・アプリケーションのコンポーネントである Server オブジェクト（「第 1 章の 2 ページ「サーバ・アプリケーション用の CORBA オブジェクトのインプリメンテーション」」を参照）でインプリメントされます。

`Server::create_servant()` オペレーションにより、CORBA オブジェクト・インプリメンテーションのインスタンスがサーバ・マシンのメモリにマップされます。このオブジェクト・インプリメンテーションのインスタンスを、オブジェクトのサーバントと呼びます。正式には、サーバントとはアプリケーションの OMG IDL ファイルに定義されるインターフェイスをインプリメントする C++ クラスのインスタンスです。サーバントは、`Server::create_servant()` オペレーションに記述する C++ `new` 文によって生成されます。

オブジェクトのサーバントが作成されたら、TP フレームワークはそのサーバントの `Tobj_ServantBase::activate_object()` オペレーションを呼び出します。`Tobj_ServantBase::activate_object()` オペレーションは、すべてのオブジェクト・インプリメンテーション・クラスが継承する `Tobj_ServantBase` 基本クラスで定義される仮想オペレーションです。TP フレームワークは、`Tobj_ServantBase::activate_object()` オペレーションを呼び出してサーバントをオブジェクト ID (OID) に関連付けます。反対に、TP フレームワークが `Tobj_ServantBase::deactivate_object()` オペレーションを呼び出すと、サーバントと OID との関連付けが解除されます。

オブジェクトのデータがディスク上に存在し、CORBA オブジェクトを活性化するときそのデータをメモリに読み取る場合、そのオブジェクトの `Tobj_ServantBase::activate_object()` オペレーションを定義およびインプリメントすることによってそのデータを読み取ることができます。`Tobj_ServantBase::activate_object()` オペレーションには、オブジェクトの永続的な状態をメモリに読み取るために必要な特定の読み取りオペレーションを格納できます。代わりに、インプリメンテーション・ファイルにコーディングしたオブジェクトの 1 つまたは複数のオペレーションによってオブジェクトのディスク・データをメモリに読み取ることもできます。詳細については、第 1 章の 19 ページ「オブジェクトのデータの読み取りと書き込み」を参照してください。`Tobj_ServantBase::activate_object()` オペレーションの呼び出しが完了したら、オブジェクトは活性化されていると見なされます。

こうしたオブジェクトのインプリメンテーションとデータの集合から、CORBA オブジェクトの実行時のアクティブ・インスタンスが構成されます。

サーバント・プール

サーバント・プールを使用すると、CORBA サーバ・アプリケーションはサーバントと特定の OID との関連付けが解除された後でも、サーバントをメモリに格納し続けることができます。プールされたサーバントで処理できるクライアント要求が到着すると、TP フレームワークは `TP::create_servant` オペレーションをバイパスし、プールされたサーバントとクライアント要求で提供された OID の間にリンクを作成します。

このため、サーバント・プールを使用すると、CORBA サーバ・アプリケーションは、サーバントによって処理できるオブジェクト要求が到着するたびにそのサーバントを再インスタンス化するコストを最小限に抑えることができます。サーバント・プールとその使い方の詳細については、「第 2 章の 34 ページ「サーバント・プール」」を参照してください。

注記 サーバント・プールは、WebLogic Enterprise リリース 4.2 で初めて導入されました。

Server オブジェクト

Server オブジェクトは、CORBA サーバ・アプリケーション用に作成するもう 1 つのプログラミング・コード・エンティティです。Server オブジェクトは、次のタスクを実行するオペレーションをインプリメントします。

- 基本的なサーバ・アプリケーション初期化オペレーションの実行。これには、サーバ・アプリケーションによって管理されるファクトリの登録や、サーバ・アプリケーションで必要なリソースの割り当てが含まれます。サーバ・アプリケーションがトランザクションに關与する場合、Server オブジェクトは XA リソース・マネージャをオープンするコードをインプリメントします。
- クライアント要求を満たすために必要な CORBA オブジェクトのインスタンス化。
- サーバ・アプリケーションが要求の処理を完了したときのサーバ・プロセスのシャットダウンおよびクリーンアップ手順の実行。たとえば、サーバ・アプリケーションがトランザクションに關与する場合、Server オブジェクトは XA リソース・マネージャをクローズするコードをインプリメントします。

Server オブジェクトは、一般のテキスト・エディタを使用してゼロから作成します。作成した Server オブジェクトは、サーバ・アプリケーションのビルド・コマンド、`buildobjserver` への入力として提供します。Server オブジェクトの作成については、「第 2 章 BEA Tuxedo CORBA サーバ・アプリケーションの作成手順」を参照してください。

CORBA サーバ・アプリケーションの開発プロセス

この節では、次のトピックに関する背景情報を提供します。この情報は、CORBA サーバ・アプリケーションの設計およびインプリメンテーション方法に大きな影響を与えます。

- オブジェクト・リファレンスの生成
- オブジェクト状態の管理
- オブジェクトのデータの読み取りと書き込み
- デザイン・パターンの使い方

次の章に進む前にこれらのトピックに目を通しておくことは必須ではありません。ただし、ここでこれらのトピックを提供するのは、すべての CORBA サーバ・アプリケーションの基本的な設計とインプリメンテーションの問題に広く適用されるからです。

オブジェクト・リファレンスの生成

CORBA サーバ・アプリケーションの最も基本的な機能の 1 つは、クライアント・アプリケーションに対して、そのビジネス・ロジックを実行するために必要なオブジェクトのオブジェクト・リファレンスを提供することです。通常、CORBA クライアント・アプリケーションは、自身が使用する初期 CORBA オブジェクトのオブジェクト・リファレンスを次の 2 つのソースから取得します。

- Bootstrap オブジェクト
- BEA Tuxedo ドメインで管理されるファクトリ

クライアント・アプリケーションは、Bootstrap オブジェクトを使用して BEA Tuxedo ドメイン内の特定のオブジェクト・セット (FactoryFinder オブジェクトや SecurityCurrent オブジェクトなど) の初期リファレンスを解決し

1 CORBA サーバ・アプリケーションの概念

ます。Bootstrap オブジェクトについては、『BEA Tuxedo CORBA アプリケーション入門』と『BEA Tuxedo CORBA クライアント・アプリケーションの開発方法』を参照してください。

ただし、ファクトリはユーザが設計、インプリメント、および登録します。ファクトリを使用すると、クライアント・アプリケーションは CORBA サーバ・アプリケーションのオブジェクト（特に初期サーバ・アプリケーション・オブジェクト）のリファレンスを取得できます。単純に言えば、ファクトリは別の CORBA オブジェクトのオブジェクト・リファレンスを返す任意の CORBA オブジェクトです。通常、クライアント・アプリケーションは、ファクトリのオペレーションを呼び出して特定のタイプの CORBA オブジェクトのオブジェクト・リファレンスを取得します。CORBA サーバ・アプリケーションを開発する際には、ファクトリの計画とインプリメンテーションを注意深く行うことが重要です。

クライアント・アプリケーションがサーバ・アプリケーションのファクトリを見つけるしくみ

クライアント・アプリケーションは、サーバ・アプリケーションによって管理されるファクトリを FactoryFinder を介して検索できます。通常、Server オブジェクトを開発する場合、サーバ・アプリケーションによって管理されるファクトリを FactoryFinder に登録するコードを組み込みます。この登録オペレーションを通じて、FactoryFinder はサーバ・アプリケーションのファクトリを追跡し、ファクトリを要求するクライアント・アプリケーションにそれらのオブジェクト・リファレンスを提供できます。オブジェクト・リファレンスを取得するときには、ファクトリを使用してそれらを FactoryFinder に登録することをお勧めします。この方法を使用すると、クライアント・アプリケーションは CORBA サーバ・アプリケーションのオブジェクトを簡単に見つけることができます。

アクティブ・オブジェクト・リファレンスの作成

アクティブ・オブジェクト・リファレンスは、サーバ・アプリケーションがオブジェクト・リファレンスを生成するための代替りの手段を提供する機能です。アクティブ・オブジェクト・リファレンスは、一般に前節で説明したファクトリによっては作成されません。アクティブ・オブジェクト・リファレンスは、状態を持つオブジェクトを事前活性化するための手段です。オブジェクトの状態については、次の節で詳しく説明します。

既存のオブジェクト・リファレンスに関連付けられているオブジェクトは、クライアント・アプリケーションがそのオブジェクトに対して呼び出しを行うまでインスタンス化されません。ただし、アクティブ・オブジェクト・リファレンスに関連付けられているオブジェクトは、そのアクティブ・オブジェクト・リファレンスの作成時に作成され、活性化されます。アクティブ・オブジェクト・リファレンスは、イテレータ・オブジェクトなどの特定の目的に役立ちます。アクティブ・オブジェクト・リファレンスの詳細については、「第 3 章の 20 ページ「状態を持つオブジェクトの事前活性化」」を参照してください。

注記 アクティブ・オブジェクト・リファレンスの機能は、WebLogic Enterprise バージョン 4.2 で最初に導入されました。

オブジェクト状態の管理

オブジェクト状態の管理は、大規模クライアント / サーバ・システムの重要な課題です。このようなシステムでは、スループットと応答時間の最適化が不可欠だからです。BEA Tuxedo ドメインで実行されるアプリケーションなど、高スループット・アプリケーションの大部分は、状態を持たない傾向にあります。つまり、こうしたシステムは、サービスまたはオペレーションが達成された後にメモリから状態情報をフラッシュします。

状態の管理は、CORBA ベースのサーバ・アプリケーションを記述するのに不可欠の要素です。一般に、これらのサーバ・アプリケーションの状態を、スケーラビリティと高性能を実現するような方法で管理するのは困難です。BEA Tuxedo ソフトウェアは、状態を管理すると同時に、スケーラビリティと高性能を保証するための簡単な方法を提供します。

CORBA サーバ・アプリケーションにスケーラビリティを組み込むことにより、サーバ・アプリケーションは、数百または数千のクライアント・アプリケーション、複数のマシン、複製サーバ・プロセス、およびそれに比例した多数のオブジェクトとそれらに対する呼び出しで構成される環境においても正常に機能します。

オブジェクト状態について

BEA Tuxedo ドメインでは、オブジェクト状態は、特にオブジェクトに対する複数のクライアント呼び出しにわたるそのオブジェクトのプロセス状態を表します。BEA Tuxedo ソフトウェアでは、状態を持つオブジェクトと状態を持たないオブジェクトに関する以下の定義を使用します。

オブジェクト	振る舞いの特性
状態を持たない	このオブジェクトは、自身のオペレーションの1つの呼び出しの間だけメモリにマップされ、その呼び出しが完了すると、非活性化されてそのプロセス状態がメモリからフラッシュされます。つまり、このオブジェクトの状態は、呼び出しの完了後はメモリに保持されません。
状態を持つ	このオブジェクトは、その呼び出しと呼び出しの間も活性化され、その状態はそれらの呼び出しを通して保持されます。状態は、次のような特定のイベントが発生するまでメモリに保持されます。 <ul style="list-style-type: none">■ オブジェクトが存在するサーバ・プロセスの停止またはシャットダウン■ オブジェクトが参加しているトランザクションのコミットまたはロールバック■ オブジェクトによる自身の <code>TP::deactivateEnable()</code> オペレーションの呼び出し これらのイベントについては、この節でそれぞれ詳しく説明します。

状態を持たないオブジェクトと状態を持つオブジェクトはどちらもデータを持ちますが、状態を持つオブジェクトは、それらのオペレーション呼び出しの間コンテキスト(状態)を保持するのに必要な非永続データをメモリ内に持つことができます。このため、こうした状態を持つオブジェクトの後続の呼び出しは、常に同じサーバントに送られます。反対に、状態を持たないオブジェクトの呼び出しは、BEA Tuxedo システムによって、そのオブジェクトを活性化できる任意の利用可能なサーバ・プロセスに送ることができます。

また、状態管理には、オブジェクトがアクティブであり続ける時間も含まれます。これは、サーバの性能とマシン・リソースの利用状態に重要な影響を与えます。活性化されたオブジェクトの存続期間は、オブジェクトのインターフェイスに割り当てるオブジェクトの活性化方針によって指定されます。オブジェクトの活性化方針については、次の節で説明します。

オブジェクトの状態は、クライアント・アプリケーションからは見えません。クライアント・アプリケーションは、分散オブジェクトとの既存の対話モデルをインプリメントします。クライアント・アプリケーションがオブジェクト・リファレンスを持っている限り、追加の要求に対してオブジェクトが常に利用可能であると見なされ、オブジェクトはクライアント・アプリケーションがそのオブジェクトと対話している間メモリ内に継続して保持されているように見えます。

アプリケーションの性能を最適化するには、作成するアプリケーションのオブジェクトがどのように状態を管理するのかについて注意深く計画する必要があります。オブジェクトは、非活性化される前に自身の状態を永続ストレージに保存する必要があります（該当する場合）。またオブジェクトは、活性化されるときに自身の状態を永続ストレージから復元する必要があります（該当する場合）。オブジェクト状態情報の読み取りと書き込みについては、「第 1 章の 19 ページ「オブジェクトのデータの読み取りと書き込み」」を参照してください。

注記 BEA Tuxedo リリース 8.0 では、性能の拡張として、パラレル・オブジェクトのサポートが提供されています。この機能を利用すると、特定アプリケーションのすべてのビジネス・オブジェクトを状態を持たないオブジェクトにすることができます。詳細については、『[BEA Tuxedo CORBA プログラミング・リファレンス](#)』の「第 3 章 TP フレームワーク」を参照してください。

オブジェクトの活性化方針

BEA Tuxedo ソフトウェアには、3 つのオブジェクトの活性化方針が用意されています。これらの方針をオブジェクトのインターフェイスに割り当てることで、オブジェクトがクライアント要求によって呼び出された後にメモリ内に保持される期間を指定できます。これらの方針によって、適用先のオブジェクトが一般に状態を持たないか、または状態を持つかが決定されます。

次の表に、3 つの方針とその説明を示します。

1 CORBA サーバ・アプリケーションの概念

方針	説明
Method	<p>オブジェクトは、自身のオペレーションの1つの呼び出しの間だけ活性化されます。つまり、オブジェクトは呼び出しの開始時に活性化され、呼び出しの終了時に非活性化されます。この活性化方針が適用されるオブジェクトを、メソッド・バウンド・オブジェクトと言います。</p> <p>method 活性化方針は、状態を持たないオブジェクトに関連付けられます。この活性化方針はデフォルトです。</p>
Transaction	<p>オペレーションが呼び出されたときにオブジェクトを活性化します。オブジェクトがトランザクション範囲内で活性化された場合、そのオブジェクトはトランザクションがコミットまたはロールバックされるまでは、活性化されたままです。オブジェクトがトランザクションのスキープの外で活性化された場合、その振る舞いはメソッド・バウンド・オブジェクトと同じです。この活性化方針が適用されるオブジェクトを、トランザクション・バウンド・オブジェクトと言います。</p> <p>トランザクションのスキープ内のオブジェクトの振る舞い、およびこの方針を使用するための一般的なガイドラインについては、「第 6 章 トランザクションの CORBA サーバ・アプリケーションへの統合」を参照してください。</p> <p>transaction 活性化方針は、期間が制限され、特定の環境に置かれている状態を持つオブジェクトに関連付けられます。</p>

方針	説明
Process	<p>オブジェクトは、自身のオペレーションが呼び出されたときに活性化され、次の環境下でのみ非活性化されます。</p> <ul style="list-style-type: none"> ■ このオブジェクトを管理するサーバ・プロセスがシャットダウンされた。 ■ このオブジェクトのオペレーションによって TP::deactivateEnable() オペレーションが呼び出され、その結果このオブジェクトが非活性化された。これは、アプリケーション制御の非活性化と呼ばれる BEA Tuxedo の主要機能の一部です。詳細については、第 1 章の 17 ページ「アプリケーション制御の非活性化」を参照してください。 <p>この活性化方針が適用されるオブジェクトを、プロセス・バウンド・オブジェクトと言います。process 活性化方針は、状態を持つオブジェクトに関連付けられます。</p>

オブジェクトの活性化方針を割り当てることにより、オブジェクトを非活性化させるイベントを指定できます。オブジェクトの活性化方針をオブジェクトのインターフェイスに割り当てる方法については、「第 2 章の 18 ページ「ステップ 4: オブジェクトのメモリ内での振る舞い」」を参照してください。

アプリケーション制御の非活性化

BEA Tuxedo ソフトウェアには、アプリケーション制御の非活性化という機能も用意されています。この機能を使用すると、アプリケーションは実行時にオブジェクトを非活性化できます。BEA Tuxedo ソフトウェアは、プロセス・バウンド・オブジェクトが自身に対して呼び出すことができる

TP::deactivateEnable() オペレーションを提供しています。

TP::deactivateEnable() オペレーションが呼び出されると、そのオペレーションが存在するオブジェクトは自身に対する現在のクライアント呼び出しの完了時に非活性化されます。オブジェクトは、自身に対してだけこのオペレーションを呼び出すことができます。呼び出しが行われているオブジェクト以外のオブジェクトでこのオペレーションを呼び出すことはできません。

1 CORBA サーバ・アプリケーションの概念

アプリケーション制御の非活性化機能は、オブジェクトに対する一定数のクライアント呼び出しの間だけそのオブジェクトをメモリに保持し、クライアント・アプリケーションからオブジェクトにそのオブジェクトの処理が終了したことを通知できるようにする場合に特に役立ちます。その時点で、オブジェクトは自身をメモリから消去します。

したがって、アプリケーション制御の非活性化を使用すると、オブジェクトはプロセス・バウンド・オブジェクトと同じようにメモリに保持されます。つまり、オブジェクトは自身に対するクライアント呼び出しによって活性化され、その初期クライアント呼び出しの完了後もメモリに保持されます。このオブジェクトは、それが存在するサーバ・プロセスをシャットダウンせずに非活性化できます。

アプリケーション制御の非活性化の代わりに方法は、トランザクションをスコープしてクライアント・アプリケーションとオブジェクト間の会話を管理することです。ただし、トランザクションは本質的にコストが高く、一般にその期間が不定の状態では適していません。

一般に、アプリケーション制御の非活性化または会話のトランザクションを選択する場合、それらにディスク書き込みオペレーションが含まれているかどうかを調べます。会話に読み取り専用オペレーションが含まれているか、またはメモリ内のみの状態の管理が含まれている場合、アプリケーション制御の非活性化の方が適しています。会話中または会話の最後にデータをディスクに書き込む場合は、トランザクションの方が適しています。

注記 アプリケーション制御の非活性化を使用して、クライアント・アプリケーションとサーバ・アプリケーションによって管理されるオブジェクト間の会話モデルをインプリメントする場合、オブジェクトは最終的に `TP::deactivateEnable()` オペレーションを呼び出す必要があります。呼び出さない場合、オブジェクトはメモリ内で永久的にアイドル状態に置かれます。これは、`TP::deactivateEnable()` オペレーションの呼び出し前にクライアント・アプリケーションがクラッシュした場合に危険を招くおそれがあります。一方、トランザクションはタイムアウト・メカニズムをインプリメントして、オブジェクトが永久的にアイドル状態に置かれることを防ぎます。これは、2つの会話モデルのいずれかを選択する際のもう1つの考慮事項です。

アプリケーション制御の非活性化は、次の手順を使用してオブジェクトにインプリメントします。

1. インプリメンテーション・ファイルで、アプリケーション制御の非活性化を使用するインターフェイスのオペレーション内の適切な位置に `TP::deactivateEnable()` オペレーションの呼び出しを挿入します。
2. インプリメンテーション・コンフィギュレーション・ファイル (ICF ファイル) で、`process` 活性化方針を、`TP::deactivateEnable()` オペレーションを呼び出すオペレーションが含まれるインターフェイスに割り当てます。
3. 「第 2 章の 22 ページ「ステップ 5: サーバ・アプリケーションのコンパイルとリンク」と「第 2 章の 23 ページ「ステップ 6: サーバ・アプリケーションのデプロイ」」で説明するとおり、アプリケーションをビルドおよびデプロイします。

オブジェクトのデータの読み取りと書き込み

サーバ・アプリケーションによって管理される CORBA オブジェクトの多くは、外部ストレージにデータを持っています。こうした外部に格納されているデータは、オブジェクトの永続状態と見なされます。オブジェクト状態管理が適切に機能するには、オブジェクト・インプリメンテーションの適切なポイントで永続状態を処理する必要があります。

オブジェクトの永続状態の読み取りと書き込みに関するクライアント/サーバ・アプリケーションのさまざまな要件のために、TP フレームワークはディスク上の永続オブジェクト状態を自動的に処理できません。一般に、オブジェクトの永続状態が 1 つまたは複数のクライアント呼び出しによって修正される場合、そのオブジェクトが非活性化される前に永続状態を保存する必要があります。また、オブジェクトが活性化されている間にどのようにオブジェクトのデータを格納または初期化するかを注意深く計画する必要があります。

以降の節では、オブジェクトの永続状態を処理するためのメカニズムについて説明し、特定の環境でオブジェクト状態を読み書きする方法についての一般的なヒントを示します。具体的には、次のトピックについて説明します。

- オブジェクトの永続状態を読み書きするためのメカニズム
- オブジェクトの活性化時の状態の読み取り
- オブジェクトの個々のオペレーションでの状態の読み取り

1 CORBA サーバ・アプリケーションの概念

- 状態を持たないオブジェクトと永続状態
- 状態を持つオブジェクトと永続状態
- オブジェクトの非活性化の責任
- 不必要な I/O の回避

永続状態の読み書きの選択方法は、クライアント / サーバ・アプリケーションの要件、特にデータの構築方法によって異なります。一般に、最も重要なのは、特に XA リソース・マネージャが関与する場合に、ディスク・オペレーションの数を最小限に抑えることです。

オブジェクトの永続状態を読み書きするためのメカニズム

表 1-1 と表 1-2 に、オブジェクトの永続状態を読み書きするためのメカニズムを示します。

表 1-1 オブジェクトの永続状態を読み取るためのメカニズム

メカニズム	説明
Tobj_ServantBase:: activate_object()	<p>オブジェクトのサーバントが作成されたら、TP フレームワークはそのサーバントの</p> <p>Tobj_ServantBase::activate_object() オペレーションを呼び出します。第 1 章の 8 ページ「CORBA オブジェクトの実行時のインスタンス化」で説明したとおり、このオペレーションはクライアント/サーバ・アプリケーション用に定義するすべての CORBA オブジェクトが継承する Tobj_ServantBase 基本クラスで定義されます。</p> <p>オブジェクトの</p> <p>Tobj_ServantBase::activate_object() オペレーションを定義およびインプリメントしないことを選択することも可能です。この場合、TP フレームワークがオブジェクトを活性化したときに特定のオブジェクト状態の処理に関しては何も発生しません。ただし、このオペレーションを定義およびインプリメントする場合、オブジェクトの永続状態の一部または全部をメモリに読み取るコードをオペレーションに組み込むことができます。このため、</p> <p>Tobj_ServantBase::activate_object() オペレーションを使用すると、サーバ・アプリケーションには、オブジェクトの永続状態をメモリに読み取るための最初の機会が与えられます。</p> <p>オブジェクトの OID にデータベース・キーが含まれている場合、</p> <p>Tobj_ServantBase::activate_object() オペレーションはそのキーを OID から抽出するための手段だけを提供します。</p> <p>Tobj_ServantBase::activate_object() オペレーションのインプリメントについては、「第 2 章の 8 ページ「ステップ 2: 各インターフェイスのオペレーションをインプリメントするメソッドの記述」を参照してください。Tobj_ServantBase::activate_object() オペレーションのインプリメントの例については、「第 3 章 Basic CORBA サーバ・アプリケーションの設計とインプリメント」を参照してください。</p>

1 CORBA サーバ・アプリケーションの概念

メカニズム	説明
オブジェクトのオペレーション	オブジェクトで定義する個々のオペレーションの内部に、オブジェクトの永続状態を読み取るコードを組み込むことができます。

表 1-2 オブジェクトの永続状態を読み取るためのメカニズム

メカニズム	説明
<code>Tobj_ServantBase::deactivate_object()</code>	<p>オブジェクトが TP フレームワークによって非活性化される場合、TP フレームワークはオブジェクト非活性化の最終ステップとしてこのオペレーションを呼び出します。<code>Tobj_ServantBase::activate_object()</code> オペレーションと同様、<code>Tobj_ServantBase::deactivate_object()</code> オペレーションも <code>Tobj_ServantBase</code> クラスで定義されます。オブジェクトの <code>deactivate_object()</code> オペレーションは、メモリからフラッシュするか、またはデータベースに書き込む特定のオブジェクト状態が存在する場合にオプションでインプリメントします。</p> <p><code>Tobj_ServantBase::deactivate_object()</code> オペレーションは、サーバ・アプリケーションに、オブジェクトの非活性化前に永続状態をディスクに書き込む最後の機会を提供します。</p> <p>オブジェクトがメモリ内のデータを保持するか、任意の目的のためにメモリを割り当てる場合、<code>Tobj_ServantBase::deactivate_object()</code> オペレーションをインプリメントしてオブジェクトが最後にメモリからデータをフラッシュできるようにします。オブジェクトの非活性化前にメモリから状態をフラッシュすることは、メモリ・リークを回避するために不可欠です。</p>

1 CORBA サーバ・アプリケーションの概念

メカニズム	説明
オブジェクトのオペレーション	<p>ディスクから永続状態を読み取る個々のオペレーションをオブジェクト上で持つことができるように、永続状態をディスクに書き込む個々のオペレーションを持つことができます。</p> <p>一般に、メソッド・バウンド・オブジェクトとプロセス・バウンド・オブジェクトの場合、データベース書き込みオペレーションはこれらのオペレーションの中で実行し、 <code>Tobj_ServantBase::deactivate_object()</code> オペレーションでは実行しません。</p> <p>ただし、トランザクション・バウンド・オブジェクトの場合、 <code>Tobj_ServantBase::deactivate_object()</code> オペレーションで永続状態を書き込むと、トランザクション・サーバ・アプリケーションにとって意味のあるオブジェクト管理上の効率化が実現されます。</p>

注記 `Tobj_ServantBase::deactivate_object()` オペレーションを使用して永続状態をディスクに書き込む場合、ディスクへの書き込み中に発生するエラーはクライアント・アプリケーションにはレポートされません。このため、このオペレーションでデータをディスクに書き込むのは、オブジェクトがトランザクション・バウンドの場合 (`transaction` 活性化方針が割り当てられている場合) か、または `TransactionCurrent` オブジェクトを呼び出すことによってトランザクション内でディスク書き込みオペレーションをスコープする場合です。トランザクション内でディスクへの書き込み中に発生したエラーは、クライアント・アプリケーションにレポートできます。

`Tobj_ServantBase::deactivate_object()` オペレーションを使用してオブジェクト状態をディスクに書き込む方法については、「第 2 章の 33 ページ「`Tobj_ServantBase::deactivate_object()`」での状態処理に関する注意」を参照してください。

オブジェクトの活性化時の状態の読み取り

オブジェクトの `Tobj_ServantBase::activate_object()` オペレーションを使用した永続状態の読み取りは、次の条件のいずれかが存在するときに適しています。

- オブジェクトのすべてのオペレーションでオブジェクト・データが常に使用または更新される。
- オブジェクトのすべてのデータを 1 つのオペレーションで読み取ることができる。

`Tobj_ServantBase::activate_object()` オペレーションを使用した永続状態の読み取りには、次のメリットがあります。

- オブジェクト・データを使用する各オペレーションでコードを複製する代わりに、データを読み取るコードを 1 度記述するだけで済みます。
- オブジェクトのデータを使用するオペレーションは、そのデータの読み取りを実行する必要がありません。このため、状態の初期化とは独立した方法でオペレーションを記述できます。

オブジェクトの個々のオペレーションでの状態の読み取り

活性化方針に関係なく、どのオブジェクトを使用する場合も、そのデータを必要とする各オペレーションで永続データを読み取ることができます。つまり、永続状態の読み取りを `Tobj_ServantBase::activate_object()` オペレーションの外部で処理できます。この方法は、次の場合に適しています。

- オブジェクト状態が、複数のオペレーションで読み取りまたは書き込みを行う必要がある個別のデータ・エレメントで構成されている。
- オブジェクトがその活性化時に必ずしも状態データを使用または更新しない。

一例として、顧客の投資ポートフォリオを表すオブジェクトを考えてみましょう。このオブジェクトには、各投資に対応する複数の個別レコードが含まれているとします。特定のオペレーションがポートフォリオの1つの投資だけに影響を与える場合、そのオペレーションを使用して1つのレコードを読み取る方が、オブジェクトを呼び出すたびに投資ポートフォリオ全体を自動的に読み取る汎用の `Tobj_ServantBase::activate_object()` オペレーションを使用するより効率的です。

状態を持たないオブジェクトと永続状態

状態を持たないオブジェクト、つまり、`method` 活性化方針で定義されたオブジェクトの場合、次のことを保証する必要があります。

- 要求によって必要とされる永続状態が、オペレーションのビジネス・ロジックが実行されるまでにメモリに読み込まれること。
- 永続状態に対する変更が、呼び出しの終了までに外部に書き込まれること。

TP フレームワークは、活性化時にオブジェクトの

`Tobj_ServantBase::activate_object()` オペレーションを呼び出します。オブジェクトが自身のディスク上の永続状態へのキーを含む OID を持っている場合、`Tobj_ServantBase::activate_object()` オペレーションはそのキーを OID から抽出するための唯一の機会をオブジェクトに提供します。

状態を持たないオブジェクトがトランザクションに参加できるようにする場合、一般に、そのオブジェクトの個々のメソッドの中でそのオブジェクトが永続状態をディスクに書き込むようにします。ただし、状態を持たないオブ

ジェクトが常にトランザクションに關与する（このオブジェクトが呼び出されたときにトランザクションが常にスコープされる）場合、

`Tobj_ServantBase::deactivate_object()` オペレーションでデータベース書き込みオペレーションを処理することを選択できます。これは、データベース書き込みオペレーションを正確にコミットまたはロールバックするための信頼できるメカニズムが XA リソース・マネージャに用意されているからです。

注記 オブジェクトがメソッド・バウンドの場合でも、2つのサーバ・プロセスが同じディスク・データに同時にアクセスする可能性を考慮に入れる必要があります。このような場合、並行性管理テクニックを使用できます。その最も簡単な方法は、トランザクションです。トランザクションとトランザクション・オブジェクトの詳細については、「第 6 章 トランザクションの CORBA サーバ・アプリケーションへの統合」を参照してください。

サーバント・プールと状態を持たないオブジェクト

サーバント・プールは、状態を持たないオブジェクトに対して特に便利な機能です。CORBA サーバ・アプリケーションがサーバントをプールすると、クライアントがオブジェクトを呼び出すたびにそのオブジェクトをインスタンス化するコストを大幅に低減できます。「第 1 章の 9 ページ「サーバント・プール」」の節で説明したとおり、プールされたサーバントはそのクライアント呼び出しの完了後もメモリに保持されます。アプリケーションで特定のオブジェクトを繰り返し呼び出す可能性がある場合、サーバントをプールすると、各クライアントに対してオブジェクトのメソッドではなくデータだけをメモリに読み取り、メモリから書き込むだけで済みます。オブジェクトのメソッドをメモリに読み取るコストが高い場合、サーバント・プールによってそのコストを低減できます。

サーバント・プールをインプリメントする方法については、「第 2 章の 34 ページ「サーバント・プール」」を参照してください。

状態を持つオブジェクトと永続状態

状態を持つオブジェクトの場合、必要な場合にのみ永続状態を読み書きする必要があります。この場合、次の最適化が必要になる場合があります。

- プロセス・バウンド・オブジェクトの場合、オブジェクトが長期間にわたって大量のメモリを割り当てる状態を回避します。

1 CORBA サーバ・アプリケーションの概念

- トランザクション・バウンド・オブジェクトの場合、トランザクションの結果がわかっているときには、
`Tobj_ServantBase::deactivate_object()` オペレーションが呼び出されるまで永続状態の書き込みを延期できます。

一般に、トランザクション・バウンド・オブジェクトは、すべてのデータベース書き込みまたはロールバック・オペレーションを自動的に処理するXA リソース・マネージャに依存する必要があります。

注記 トランザクションに關与するオブジェクトの場合、XA リソース・マネージャによって管理されない外部ストレージへのオブジェクトのデータ書き込みはサポートされません。

オブジェクトとトランザクションの詳細については、「第 6 章 トランザクションの CORBA サーバ・アプリケーションへの統合」を参照してください。

サーバント・プールと状態を持つオブジェクト

サーバント・プールは、プロセス・バウンド・オブジェクトに対しては意味を持ちません。ただし、アプリケーション設計によっては、サーバント・プールによってトランザクション・バウンド・オブジェクトの性能が向上する場合があります。

オブジェクトの非活性化の責任

前節で説明したとおり、`Tobj_ServantBase::deactivate_object()` オペレーションは、オブジェクトの永続状態をディスクに書き込むための手段としてインプリメントします。また、このオペレーションは、保持されているオブジェクト・データをメモリからフラッシュして、オブジェクトのサーバントでそのオブジェクトの別のインスタンスを活性化できるようにするためにも使用されます。ただし、オブジェクトの

`Tobj_ServantBase::deactivate_object()` オペレーションの呼び出しによってそのオブジェクトのデストラクタも呼び出されると見なしてはなりません。

不必要な I/O の回避

オブジェクトで不必要な I/O を行うことによってアプリケーションの効率を下げないように注意する必要があります。注意が必要な状態は以下のとおりです。

- オブジェクトの多くのオペレーションがオブジェクト状態を使用せず、オブジェクト状態に影響を与えない場合、これらのオペレーションを呼び出すたびに状態を読み書きするのは非効率です。設計を修正して、これらのオブジェクトが状態を必要とするオペレーションでのみ状態を処理するようにします。この場合、オブジェクトの活性化でそのすべての永続状態が読み取られないようにすることができます。
- オブジェクト状態が複数のオペレーションで読み込まれるデータで構成されている場合、次のいずれかを行うことによって、オブジェクトの呼び出し時に必要なオペレーションだけを行うようにします。

- すべてのオペレーションに共通の状態だけを

`Tobj_ServantBase::activate_object()` オペレーションで読み取ります。それ以外の状態の読み取りは、それらを必要とするオペレーションのみで行うようにします。

- 変更された状態だけを書き込みます。そのためには、活性化中に変更されたデータを示すフラグを管理するか、ビフォー・データ・イメージとアフター・データ・イメージを比較します。

一般的な最適化の方法は、活性化時に `dirtyState` フラグを初期化し、オブジェクトの活性化中にこのフラグが変更された場合にだけ

`Tobj_ServantBase::deactivate_object()` オペレーションでデータを書き込むことです。ただし、これはオブジェクトが常に同じサーバント・プロセスで活性化されると保証できる場合にだけ使用できます。

活性化のサンプル

オブジェクトが活性化されるときに行われる一連のアクティビティの例については、『BEA Tuxedo CORBA アプリケーション入門』を参照してください。

デザイン・パターンの使い方

優れた設計に基づいてアプリケーションのビジネス・ロジックを構築することは重要です。BEA Tuxedo ソフトウェアには、このニーズを満たすデザイン・パターンのセットが用意されています。デザイン・パターンは、特定の設計問題に対する構造化されたソリューションです。デザイン・パターンの価値は、再利用およびほかのデザイン問題への適用が可能な形式で表現できることにあります。

BEA Tuxedo デザイン・パターンは、エンタープライズ・クラスのアプリケーション設計の問題に対する構造化されたソリューションです。これらを使用すると、大規模クライアント / サーバ・アプリケーションを適切に設計できます。

ここで説明するデザイン・パターンは、CORBA クライアント・アプリケーションおよびサーバ・アプリケーションで優れた設計手法を採用するためのガイドです。これらのデザイン・パターンは CORBA クライアント・アプリケーションおよびサーバ・アプリケーションの設計の重要な部分です。このマニュアルの各章には、これらのデザイン・パターンを使用して University サンプル・アプリケーションをインプリメントする方法が示されています。

Process-Entity デザイン・パターン

Process-Entity デザイン・パターンは、大規模なエンタープライズ・クラスのクライアント / サーバ・アプリケーションに適用されます。このデザイン・パターンは、『Object-Oriented Design Patterns』(Gamma ほか著)では Flyweight パターンと呼ばれ、ほかの出版物ではモデル / ビュー / コントローラと呼ばれています。

このパターンでは、クライアント・アプリケーションは存続期間の長いプロセス・オブジェクトを作成し、このオブジェクトと対話して要求を行います。たとえば、University サンプル・アプリケーションでは、このオブジェクトはクライアント・アプリケーションに代わってコース参照オペレーションを処理する登録係です。コース自体はデータベース・エンティティで、クライアント・アプリケーションからは見えません。

Process-Entity デザイン・パターンのメリットは次のとおりです。

- 細粒度オブジェクトをインプリメントせずに細粒度オブジェクトのメモリットを実現できます。代わりに CORBA `struct` データ型を使用してオブジェクトをシミュレートします。
- 1つのオブジェクト(プロセス・オブジェクト)だけがメモリにマップされるので、マシンのリソース利用が最適化されます。対照的に、大規模なデプロイメントでは、各データベース・エンティティが独立したオブジェクト・インスタンスとしてメモリに活性化される場合、処理する必要があるオブジェクトの数によってマシンのリソースが急激に消費されてしまいます。
- これらはクライアント・アプリケーションからは見えないので、データベース・エンティティを CORBA オブジェクトとしてインプリメントする必要がありません。代わりに、エンティティはサーバ・プロセス内のローカル言語オブジェクトとしてインプリメントできます。これは、3層設計の基本的な原則です。ただし、これは多くのビジネスの運用方法を正確にモデル化します(現実の大学における登録係など)。大学で登録係を務める1人の人間は、複数の学生のために大規模なコース・データベースを処理できます。このため、学生ごとに登録係は必要ありません。このように、プロセス・オブジェクトの状態は、エンティティ・オブジェクトの状態とは区別されます。

Process-Entity デザイン・パターンの適用例については、「第 3 章 Basic CORBA サーバ・アプリケーションの設計とインプリメント」を参照してください。Process-Entity デザイン・パターンの詳細については、『CORBA 技術情報』を参照してください。

List-Enumerator デザイン・パターン

List-Enumerator デザイン・パターンも、大規模なエンタープライズ・クラスのクライアント / サーバ・アプリケーションに適用されます。

List-Enumerator デザイン・パターンは、BEA Tuxedo の主要機能であるアプリケーション制御のオブジェクト活性化を利用して、複数のクライアント呼び出しの間メモリ内で保持および追跡されるデータのキャッシュを処理し、不要になったときにそのデータをメモリからフラッシュします。

List-Enumerator デザイン・パターンの適用例については、「第 3 章 Basic CORBA サーバ・アプリケーションの設計とインプリメント」を参照してください。

1 CORBA サーバ・アプリケーションの概念

List-Enumerator デザインをインプリメントするのに特に役立つツール、オブジェクトの事前活性化については、「第 3 章の 20 ページ「状態を持つオブジェクトの事前活性化」」を参照してください。

2 BEA Tuxedo CORBA サーバ・ア プリケーションの作 成手順

この章では、CORBA サーバ・アプリケーションを作成するための手順について説明します。この章で説明する手順は、決定的なものではありません。サーバ・アプリケーションによっては、ほかの手順を行う必要があります。また、これらの手順のいくつかについては順番を変更することができます。ただし、これらの手順は、すべての CORBA サーバ・アプリケーションの開発プロセスに共通のものであります。

ここでは、以下の内容について説明します。

- CORBA サーバ・アプリケーション開発プロセスの概要
- 開発とデバッグのヒント
- サーバント・プール
- デレゲーション・ベースのインターフェイス・インプリメンテーション

この章では、最初に手順の要約と、このマニュアルで使用する開発ツールおよび開発コマンドのリストを示します。デプロイメント環境によっては、ほかのソフトウェア開発ツールも使用します。このため、この章で説明するツールとコマンドも決定的なものではありません。

この章では、BEA Tuxedo ソフトウェアに付属の Basic University サンプル・アプリケーションの中の例を使用します。Basic University サンプル・アプリケーションの詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。このマニュアルで使用するツールとコマンドの詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

マルチスレッド CORBA サーバ・アプリケーションの作成については、「第 4 章 マルチスレッド CORBA サーバ・アプリケーションの作成」を参照してください。相互参照が当たっていません (2003/02/27)

CORBA サーバ・アプリケーション開発プロセスの概要

サーバ・アプリケーションを作成するための基本的な手順は次のとおりです。

ステップ 1: サーバ・アプリケーション用の OMG IDL ファイルのコンパイル
相互参照が当たっていません (2003/02/27)

ステップ 2: 各インターフェイスのオペレーションをインプリメントするメソッドの記述

ステップ 3: サーバ・オブジェクトの作成

ステップ 4: オブジェクトのメモリ内での振る舞い

ステップ 5: サーバ・アプリケーションのコンパイルとリンク

ステップ 6: サーバ・アプリケーションのデプロイ

BEA Tuxedo ソフトウェアには、次の開発ツールと開発コマンドが用意されています。

ツール	説明
IDL コンパイラ	アプリケーションの OMG IDL ファイルをコンパイルします。

ツール	説明
genicf	インプリメンテーション・コンフィギュレーション・ファイル (ICF ファイル) を生成します。このファイルを修正して、非デフォルト・オブジェクトの活性化方針とトランザクション方針を指定できます。
buildobjserver	CORBA サーバ・アプリケーションの実行可能イメージを作成できます。
tmloadcf	TUXCONFIG ファイルを作成します。これは、サーバ・アプリケーションのコンフィギュレーションを指定する CORBA ドメイン用のバイナリ・ファイルです。
tmadmin	特に、トランザクション・アクティビティのログを作成します。これは、いくつかのサンプル・アプリケーションで使用されます。

ステップ 1: サーバ・アプリケーション用の OMG IDL ファイルのコンパイル

BEA Tuxedo ドメインで実行されるアプリケーションのクライアントおよびサーバ部分の基本構造は、そのアプリケーションの OMG IDL ファイルに記載される文によって決定されます。アプリケーションの OMG IDL ファイルをコンパイルする場合、IDL コンパイラは、`idl` コマンドで指定するオプションに応じて、次の図に示すファイルの一部または全部を生成します。影付きのコンポーネントは、サーバ・アプリケーションを作成するために修正するファイルです。

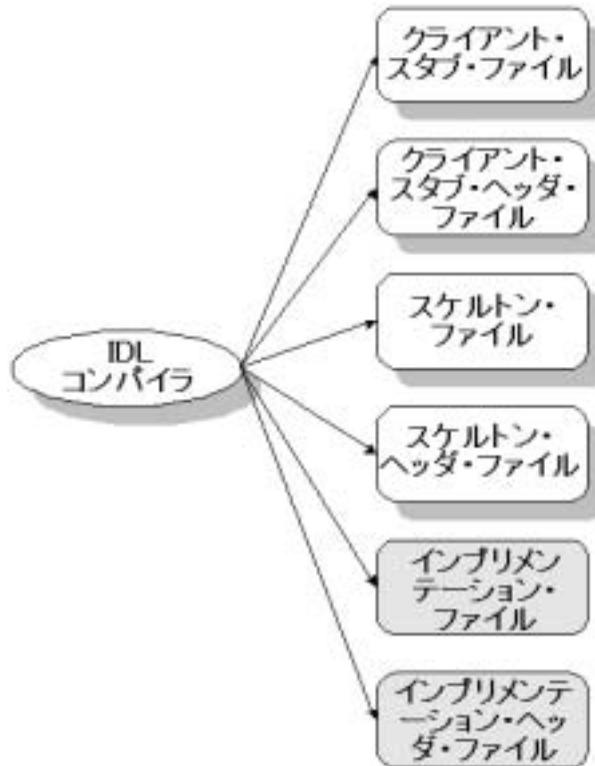


表 に、IDL コンパイラによって生成されるファイルを示します。

表 2-1 IDL コンパイラによって生成されるファイル

ファイル	デフォルト名	説明
クライアント・スタブ・ファイル	<code>application_c.cpp</code>	要求を送信するために生成されたコードを格納します。
クライアント・スタブ・ヘッダ・ファイル	<code>application_c.h</code>	OMG IDL ファイルで指定された各インターフェイスと型のクラス定義を格納します。

ステップ 1: サーバ・アプリケーション用の OMG IDL ファイルのコンパイル

表 2-1 IDL コンパイラによって生成されるファイル

ファイル	デフォルト名	説明
スケルトン・ファイル	<code>application_s.cpp</code>	OMG IDL ファイルで指定した各インターフェイス用のスケルトンが含まれます。スケルトンは、実行時にクライアント要求をサーバ・アプリケーション内の適切なオペレーションにマップします。
スケルトン・ヘッダ・ファイル	<code>application_s.h</code>	スケルトンのクラス定義を格納します。
インプリメンテーション・ファイル	<code>application_i.cpp</code>	OMG IDL ファイルで指定されたインターフェイスのオペレーションをインプリメントするメソッドのシグニチャを格納します。
インプリメンテーション・ヘッダ・ファイル	<code>application_i.h</code>	OMG IDL ファイルで指定された各インターフェイスの初期クラス定義を格納します。

IDL コンパイラの使い方

表 2-1 に示したファイルを生成するには、次のコマンドを入力します。相互参照が当たっていません (2003/02/27)

```
idl [options] idl-filename [icf-filename]
```

idl コマンド構文のパラメータは次のとおりです。

- `options` は、IDL コンパイラへの 1 つまたは複数のコマンド行オプションを表します。コマンド行オプションについては、『[BEA Tuxedo CORBA プログラミング・リファレンス](#)』を参照してください。インプリメンテーション・ファイルを生成する場合は、`-i` オプションを指定する必要があります。
- `idl-filename` は、アプリケーションの OMG IDL ファイルの名前を表します。
- `icf-filename` は、アプリケーションのインプリメンテーション・コンフィギュレーション・ファイル (ICF ファイル) の名前を表すオプションのパラメータです。ICF ファイルは、オブジェクトの活性化方針を指定

するか、または生成するスケルトンおよびインプリメンテーション・ファイルのインターフェイスの数を制限するために使用します。ICF ファイルの使い方については、ページ 2-14 の「ステップ 4: オブジェクトのメモリ内での振る舞い」を参照してください。相互参照が当たっていません (2003/02/27)

注記 WebLogic Enterprise 5.1 では、プラグマの C++ IDL コンパイラ・インプリメンテーションが変更され、CORBA 2.3 機能がサポートされるようになりました。これにより、IDL ファイルが影響を受ける場合があります。CORBA 2.3 機能は、プラグマ接頭辞定義が影響を及ぼす可能性があるスコープを変更します。プラグマはインクルードされる IDL ファイルに含まれる定義には影響を及ぼさず、またインクルードされる IDL ファイル内で行われるプラグマ接頭辞定義はそれらのファイルの外部のオブジェクトに影響を及ぼしません。

C++ IDL コンパイラは、プラグマ接頭辞の処理を訂正するよう変更されました。この変更はオブジェクトのリポジトリ ID に影響を与えるため、`_narrow` などのオペレーションでエラーが発生する場合があります。

このようなエラーを防ぐには、次のことを行います。

- IDL をリポジトリにリロードする場合、アプリケーションのクライアント・スタブとサーバ・スケルトンを再生成する必要があります。
- 任意のクライアント・スタブまたはサーバ・スケルトンを再生成する場合、アプリケーションのすべてのスタブとスケルトンを再生成し、IDL をインターフェイス・リポジトリにリロードする必要があります。

IDL コンパイラと `idl` コマンドの詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

スケルトンとインプリメンテーション・ファイルの生成

次のコマンド行は、OMG IDL ファイル `univb.idl` のクライアント・スタブ・ファイル、スケルトン・ファイル、初期インプリメンテーション・ファイル、スケルトン・ヘッダ・ファイル、およびインプリメンテーション・ヘッダ・ファイルを生成します。

```
idl -i univb.idl
```

`idl` コマンドの詳細については、『[BEA Tuxedo CORBA プログラミング・リファレンス](#)』を参照してください。BEA Tuxedo University サンプル・アプリケーション用のこれらのファイルを生成する方法については、『[BEA Tuxedo CORBA University サンプル・アプリケーション](#)』を参照してください。

注記 非デフォルトのオブジェクトの活性化方針またはトランザクション方針を指定する場合、または生成するスケルトン・ファイルとインプリメンテーション・ファイルのインターフェイスの数を制限する場合、インプリメンテーション・コンフィギュレーション・ファイル (ICF) を生成し、この ICF ファイルを IDL コンパイラに受け渡す必要があります。詳細については、ページ 2-14 の「ICF でオブジェクトの活性化方針とトランザクション方針を指定する方法」を参照してください。相互参照が当たっていません (2003/02/27)

tie クラスの生成

IDL コンパイラには、インターフェイスの `tie` クラス・テンプレートを生成するために使用できる `-T` コマンド行オプションも用意されています。CORBA アプリケーションの `tie` クラスのインプリメントについては、ページ 2-29 の「デレゲーション・ベースのインターフェイス・インプリメンテーション」を参照してください。

ステップ 2: 各インターフェイスのオペレーションをインプリメントするメソッドの記述

サーバ・アプリケーション・プログラマは、アプリケーションの OMG IDL ファイルで定義した各インターフェイスのオペレーションをインプリメントするメソッドを記述します。

インプリメンテーション・ファイルには以下のものが含まれます。

- OMG IDL ファイルで指定された各オペレーションのメソッド宣言
- アプリケーションのビジネス・ロジック、インクルード・ファイル、およびアプリケーションで使用するその他のデータ
- 各インターフェイス・インプリメンテーションのコンストラクタとデストラクタ (これらのインプリメントはオプション)
- オプションで、`Tobj_ServantBase::activate_object()` オペレーションと `Tobj_ServantBase::deactivate_object()` オペレーション

`Tobj_ServantBase::activate_object()` オペレーションと

`Tobj_ServantBase::deactivate_object()` オペレーションの中で、オブジェクトの活性化または非活性化に関連する特定の手順を実行するコードを記述します。これには、オブジェクトの永続状態のディスクからの読み取りとディスクへの書き込みがそれぞれ含まれます。これらのオペレーションをオブジェクトにインプリメントする場合、インプリメンテーション・ヘッダ・ファイルを編集して、これらのオペレーションを使用する各インプリメンテーションにそれらの定義を追加する必要があります。

IDL コンパイラによって生成されるインプリメンテーション・ファイル

サーバ・アプリケーションのインプリメンテーション・ファイルはすべて手動で作成できますが、インプリメンテーション・ファイルを記述する出発点として、IDL コンパイラが生成するインプリメンテーション・ファイルを使用することもできます。IDL コンパイラによって生成されるインプリメンテーション・ファイルには、アプリケーションのインターフェイス用に定義された各オペレーションをインプリメントするメソッドのシグニチャが含まれます。

通常、このインプリメンテーション・ファイルは、IDL コンパイラを呼び出すコマンドで `-i` オプションを指定することによって 1 度だけ作成します。アプリケーションのインターフェイスを繰り返し修正し、これらのインターフェイスのオペレーション（オペレーション・シグニチャを含む）を変更した場合は、必要なすべての変更をインプリメンテーション・ファイルに追加してこれらの変更を反映させる必要があります。

ファクトリのインプリメント

ページ 1-4 の「クライアント・アプリケーションがサーバ・アプリケーションの CORBA オブジェクトをアクセスおよび操作する方法」で説明したように、クライアント・アプリケーションがサーバ・アプリケーションによって管理されるオブジェクトを簡単に検索できるようにするためには、ファクトリを作成する必要があります。相互参照が当たっていません (2003/02/27) ファクトリはインプリメントするほかの CORBA オブジェクトに似ていますが、FactoryFinder オブジェクトに登録する必要があります。ファクトリの登録については、ページ 2-10 の「ファクトリを作成および登録するコードの記述」を参照してください。相互参照が当たっていません (2003/02/27)

ファクトリの主要な機能は、オブジェクト・リファレンスを作成することです。ファクトリは、`TP::create_object_reference()` オペレーションを呼び出すことによってこれを実行します。`TP::create_object_reference()` オペレーションでは、次の入力パラメータが必要となります。

- オブジェクトの OMG IDL インターフェイスのインターフェイス・リポジトリ ID
- 文字列形式のオブジェクト ID (OID)
- ルーティング基準 (オプション)

たとえば、Basic University サンプル・アプリケーションでは、RegistrarFactory インターフェイスは、次のように 1 つのオペレーションだけを指定します。

```
University::Registrar_ptr RegistrarFactory_i::find_registrar()
```

RegistrarFactory オブジェクトの find_registrar() オペレーションには、Registrar オブジェクトのリファレンスを作成するための

TP::create_object_reference() オペレーションへの次の呼び出しが含まれています。

```
CORBA::Object_var v_reg_oref =
    TP::create_object_reference(
        University::_tc_Registrar->id(),
        object_id,
        CORBA::NVlist::_nil()
    );
```

このコード・サンプルでは、次のことに留意してください。

- 次のパラメータは、Registrar オブジェクトのインターフェイス・リポジトリ ID をタイプ・コードから抽出することによってその ID を指定します。

```
University::_tc_Registrar->id()
```

- 次のパラメータは、ルーティング基準が使用されないことを指定します。この結果、Registrar オブジェクト用に作成されるオブジェクト・リファレンスは、それを作成する RegistrarFactory オブジェクトと同じグループにルーティングされます。

```
CORBA::NVlist::_nil()
```

オブジェクト・リファレンスのルーティング先のグループに影響を与えるルーティング基準の指定については、「第 8 章 BEA Tuxedo の CORBA サーバ・アプリケーションのスケーリング」を参照してください。相互参照が当たっていません (2003/02/27)

ステップ 3: サーバ・オブジェクトの作成

Server オブジェクトのインプリメントは、ほかの言語オブジェクトのインプリメントとは異なります。Server オブジェクトのヘッダ・クラスは既に作成されており、Server オブジェクト・クラスは既にインスタンス化されています。Server オブジェクトを作成するには、パッケージ済みの Server オブジェクト・クラスの特定のメソッド・セットをインプリメントします。この節では、インプリメントするこれらのメソッドについて説明します。

Server オブジェクトを作成するには、一般のテキスト・エディタで新しいファイルを作成し、次のオペレーションをインプリメントします。

オペレーション	説明
<code>Server::initialize();</code>	サーバ・アプリケーションの起動後、TP フレームワークはサーバ・アプリケーション初期化プロセスの最後のステップとしてこのオペレーションを呼び出します。このオペレーションでは、特定のサーバ・アプリケーションに必要な複数の初期化タスクを実行します。このオペレーションで提供するものについては、ページ 2-9 の「サーバ・アプリケーションの初期化」を参照してください。相互参照が当たっていません (2003/02/27)
<code>Server::create_servant();</code>	既存のサーバントで処理できないクライアント要求が送信されると、TP フレームワークはこのオペレーションを呼び出して、活性化される CORBA オブジェクトの OMG IDL インターフェイスのインターフェイス・リポジトリ ID を受け渡します。このオペレーションで提供するものについては、ページ 2-11 の「サーバントの作成」を参照してください。相互参照が当たっていません (2003/02/27)

オペレーション	説明
<code>Server::release();</code>	TP フレームワークは、サーバ・アプリケーションをシャットダウンするときこのオペレーションを呼び出します。このオペレーションには、サーバ・アプリケーションで管理されるオブジェクト・ファクトリの登録を削除し、ほかのシャットダウン・タスクを実行するためのコードが含まれます。このオペレーションで提供するものについては、ページ 2-13 の「サーバ・アプリケーションのリリース」を参照してください。相互参照が当たっていません (2003/02/27)

どのサーバ・アプリケーションにも、Server オブジェクトのインスタンスは 1 つしか存在しません。サーバ・アプリケーションが複数の CORBA オブジェクト・インプリメンテーションを管理する場合、記述する

`Server::initialize()`、`Server::create_servant()`、および `Server::release()` オペレーションにはこれらのインプリメンテーションすべてに適用するコードを組み込む必要があります。

これらのタスクの大部分のコードには、TP フレームワークとの対話が含まれます。以降の節では、これらの Server オブジェクト・オペレーションのそれぞれに対して必要なコードについて説明し、Basic University サンプル・アプリケーションのサンプル・コードを示します。

サーバ・アプリケーションの初期化

Server オブジェクトに実装する最初のオペレーションは、サーバ・アプリケーションを初期化するオペレーションです。このオペレーションは、BEA Tuxedo システムがサーバ・アプリケーションを起動するときに呼び出されます。TP フレームワークは、サーバ・アプリケーションの起動シーケンス中に Server オブジェクトの次のオペレーションを呼び出します。

```
CORBA::Boolean Server::initialize(int argc, char** argv)
```

BEA Tuxedo ドメインの `UBBCONFIG` ファイルの `SERVERS` セクションに指定する特定のサーバ・アプリケーション用の `CLOPT` パラメータは、

`Server::initialize()` オペレーションに `argc` および `argv` として受け渡されます。サーバ・アプリケーションへの引数の受け渡しについては、『BEA

[Tuxedo アプリケーション実行時の管理](#)』を参照してください。サーバ・アプリケーションに引数を受け渡す例については、『[BEA Tuxedo CORBA University サンプル・アプリケーション](#)』を参照してください。

`Server::initialize()` オペレーションの中には、該当する場合、次のことを行うコードを組み込みます。

- ファクトリの作成と登録
- マシン・リソースの割り当て
- サーバ・アプリケーションに必要なグローバル変数の初期化
- サーバ・アプリケーションによって使用されるデータベースのオープン
- XA リソース・マネージャのオープン

ファクトリを作成および登録するコードの記述

クライアント・アプリケーションがオブジェクトを簡単に検索できるようにするためのファクトリをサーバ・アプリケーションが管理する場合、そのファクトリを `FactoryFinder` オブジェクトに登録するコードを記述する必要があります。このコードは、通常サーバ・アプリケーション初期化プロセスの最後のステップとして呼び出されます。

サーバ・アプリケーションによって管理されるファクトリに登録するコードを記述するには、次のことを行います。

1. ファクトリのオブジェクト・リファレンスを作成します。

このステップでは、ページ 2-7 の「ファクトリのインプリメント」で説明したとおりオブジェクト・リファレンスを作成します。相互参照が当たっていません (2003/02/27) このステップでは、

`TP::create_object_reference()` オペレーションの呼び出しを組み込み、OMG IDL インターフェイスのインターフェイス・リポジトリ ID を指定します。次の例では、`RegistrarFactory` ファクトリのオブジェクト・リファレンス (`s_v_fact_ref` 変数で表される) が作成されます。

```
University::RegistrarFactory s_v_fact_ref =
    TP::create_object_reference(
        University::_tc_RegistrarFactory->id(),
```

2 BEA Tuxedo CORBA サーバ・アプリケーションの作成手順

```
        object_id,  
        CORBA::NVList::_nil()  
    );
```

2. ファクトリを BEA Tuxedo ドメインに登録します。

このステップでは、サーバ・アプリケーションによって管理される各ファクトリに対して次のオペレーションを呼び出します。

```
TP::register_factory (CORBA::Object_ptr factory_or,  
                    const char* factory_id);
```

TP::register_factory() オペレーションは、サーバ・アプリケーションのファクトリを FactoryFinder オブジェクトに登録します。このオペレーションでは、次の入力パラメータが必要です。

- 前述のステップ 1 で作成した、ファクトリのオブジェクト・リファレンス。
- ファクトリ・オブジェクトのインターフェイス・タイプ・コードに基づく文字列識別子。これは、ファクトリの OMG IDL インターフェイスのインターフェイス・リポジトリ ID を識別するために使用されません。

次の例では、RegistrarFactory ファクトリが BEA Tuxedo ドメインに登録されます。

```
TP::register_factory(s_v_fact_ref.in(),  
                  University::_tc_RegistrarFactory->id());
```

University::_tc_RegistrarFactory->id() パラメータに注目してください。これは、TP::create_object_reference() オペレーションで指定したパラメータと同じです。このパラメータは、オブジェクトの OMG IDL インターフェイスのインターフェイス・リポジトリ ID をそのタイプ・コードから抽出します。

サーバントの作成

サーバ・アプリケーション初期化プロセスが完了したら、サーバ・アプリケーションはクライアント要求を処理できる状態になります。CORBA オブジェクトのオペレーションに対する要求が到着し、それを処理するサーバントがメモリ内に存在しない場合、TP フレームワークは Server オブジェクトの次のオペレーションを呼び出します。

```
Tobj_Servant Server::create_servant(const char* interfaceName)
```

Server::create_servant() オペレーションには、クライアント要求によって必要とされるオブジェクトのサーバントをインスタンス化するコードが含まれます。たとえば、C++ では、このコードにはオブジェクトのインターフェイス・クラスの new 文が組み込まれます。

Server::create_servant() オペレーションでは、サーバントは OID に関連付けられません。サーバントと OID の関連付けは、TP フレームワークがそのサーバントの Tobj_ServantBase::activate_object() オペレーション(オブジェクトのインスタンス化を完了させるオペレーション)を呼び出したときに行われます。オブジェクトのコンストラクタで OID をオブジェクトに関連付けることはできません。同様に、サーバントと OID の関連付けの解除は、TP フレームワークがそのサーバントの deactivate_object() オペレーションを呼び出したときに行われます。

BEA Tuxedo システムでのこうしたサーバントの振る舞いにより、TP フレームワークは、オブジェクトの非活性化後、別のオブジェクトのインスタンス化のためにサーバントを使用できます。したがって、オブジェクトの

Tobj_ServantBase::deactivate_object() オペレーションの呼び出しによってそのオブジェクトのデストラクタも呼び出されるとは考えないでください。サーバ・アプリケーションでサーバント・プール機能を使用する場合、オブジェクトの Tobj_ServantBase::deactivate_object() オペレーションに TP::application_responsibility() オペレーションをインプリメントして、サーバントへのポインタを後で使用できるようにサーバント・プールに受け渡すことができます。サーバント・プールについては、ページ 2-28 の「サーバント・プール」を参照してください。相互参照が当たっていません (2003/02/27)

`Server::create_servant()` オペレーションでは、入力引数が1つ必要です。この引数は、サーバントを作成するためのオブジェクトの OMG IDL インターフェイスのインターフェイス リポジトリ ID を含む文字列を指定します。

このオペレーション用に記述するコードには、サーバ・アプリケーションによって管理されるオブジェクトの OMG IDL インターフェイスのインターフェイス・リポジトリ ID を指定します。実行時に、

`Server::create_servant()` オペレーションは、要求によって指定されたオブジェクトに対して必要なサーバントを返します。

次のコードは、Basic University サンプル・アプリケーションの University サーバ・アプリケーションの `Server::create_servant()` オペレーションをインプリメントします。

```
Tobj_Servant Server::create_servant(const char* intf_repos_id)
{
    if (!strcmp(intf_repos_id, University::_tc_RegistrarFactory->id())) {
        return new RegistrarFactory_i();
    }
    if (!strcmp(intf_repos_id, University::_tc_Registrar->id())) {
        return new Registrar_i();
    }
    if (!strcmp(intf_repos_id, University::_tc_CourseSynopsisEnumerator->id())) {
        return new CourseSynopsisEnumerator_i();
    }
    return 0; // unknown interface
}
```

サーバ・アプリケーションのリリース

BEA Tuxedo システム管理者が `tmsshutdown` コマンドを入力すると、TP フレームワークは、BEA Tuxedo ドメインで実行される各サーバ・アプリケーションの `Server` オブジェクトの次のオペレーションを呼び出します。

```
void Server::release()
```

`Server::release()` オペレーションでは、次のような、サーバ・アプリケーションに応じたアプリケーション固有のクリーンアップ・タスクを実行できます。

- サーバ・アプリケーションによって管理されるオブジェクト・ファクトリの登録の削除
- リソースの割り当て解除
- データベースのクローズ
- XA リソース・マネージャのクローズ

サーバ・アプリケーションがシャットダウン要求を受信すると、そのサーバ・アプリケーションはほかのリモート・オブジェクトから要求を受信することができなくなります。これは、管理者がサーバ・アプリケーションをシャットダウンする順番に影響を与えます。たとえば、あるサーバ・プロセスへの `Server::release()` オペレーションの呼び出しが別のサーバ・プロセスに含まれている場合、最初のサーバ・プロセスはシャットダウンしてはなりません。

サーバのシャットダウン中に、次の呼び出しを組み込んでサーバ・アプリケーションの各ファクトリの登録を削除できます。

```
TP::unregister_factory (CORBA::Object_ptr factory_or,  
                       const char* factory_id)
```

`TP::unregister_factory()` オペレーションの呼び出しは、`Server::release()` インプリメンテーションの最初のアクションの 1 つである必要があります。 `TP::unregister_factory()` オペレーションは、サーバ・アプリケーションのファクトリへの登録を削除します。このオペレーションでは、次の入力引数が必要です。

- ファクトリのオブジェクト・リファレンス
- ファクトリ・オブジェクトのインターフェイス・タイプ・コードに基づく文字列識別子。これは、ファクトリの OMG IDL インターフェイスのインターフェイス・リポジトリ ID を識別するために使用されます。

次の例では、Basic サンプル・アプリケーションで使用されている `RegistrarFactory` ファクトリへの登録が削除されます。

```
TP::unregister_factory(s_v_fact_ref.in(), UnivB::_tc_RegistrarFactory->id());
```

このコード例では、グローバル変数 `s_v_fact_ref` の使い方に注目してください。この変数は、`RegistrarFactory` オブジェクトを登録した `Server::initialize()` オペレーションで設定されたもので、ここで再び使用されます。

また、`UnivB::_tc_RegistrarFactory->id()` パラメータにも注目してください。これも、ファクトリの登録に使用されたインターフェイス名と同じです。

ステップ 4: オブジェクトのメモリ内での振る舞い

ページ 1-11 の「オブジェクト状態の管理」で説明したように、オブジェクトの活性化方針とトランザクション方針を割り当て、オプションでアプリケーション制御の非活性化機能を使用することによって、オブジェクトを非活性化するイベントを指定します。相互参照が当たっていません (2003/02/27)

オブジェクトの活性化方針とトランザクション方針は ICF ファイルに指定し、アプリケーション制御の非活性化は `TP::deactivateEnable()` オペレーションを介して使用します。この節では、Basic University サンプル・アプリケーションを例として使用して、これらのメカニズムをインプリメントする方法について説明します。

ここでは、次のことについて説明します。

- ICF でオブジェクトの活性化方針とトランザクション方針を指定する方法
- アプリケーション制御の非活性化をインプリメントする方法

ICF でオブジェクトの活性化方針とトランザクション方針を指定する方法

BEA Tuxedo ソフトウェアは、ページ 1-13 の「オブジェクトの活性化方針」で説明した次の活性化方針をサポートしています。相互参照が当たっていません (2003/02/27)

活性化方針	説明
method	オブジェクトのオペレーションの呼び出しの間だけオブジェクトを活性化します。
transaction	オペレーションが呼び出されたときにオブジェクトを活性化します。オブジェクトがトランザクション範囲内で活性化された場合、そのオブジェクトはトランザクションがコミットまたはロールバックされるまでは、活性化されたままです。
process	オペレーションが呼び出されたときにオブジェクトを活性化し、次のいずれかが起こったときのみオブジェクトを非活性化します。 <ul style="list-style-type: none"> ■ サーバ・アプリケーションの存在するプロセスがシャットダウンされた。 ■ オブジェクトが自身の <code>TP::deactivateEnable()</code> オペレーションを呼び出した。

BEA Tuxedo ソフトウェアは、「第 6 章 トランザクションの CORBA サーバ・アプリケーションへの統合」で説明する次のトランザクション方針もサポートしています。相互参照が当たっていません (2003/02/27)

トランザクション方針	説明
always	このオブジェクトのオペレーションが呼び出されると、この方針によって TP フレームワークはそのオブジェクトのトランザクションを開始します (活性化されているトランザクションが存在しない場合)。TP フレームワークがトランザクションを開始した場合、TP フレームワークは、オペレーションが正常に完了した場合はトランザクションをコミットし、例外が発生した場合はトランザクションをロールバックします。
optional	このオブジェクトのオペレーションが呼び出されると、この方針によって TP フレームワークはこのオブジェクトをトランザクションに組み込みます (トランザクションが活性化されている場合)。活性化されているトランザクションが存在しない場合、このオブジェクトに対して定義されている活性化方針に従ってこのオブジェクトの呼び出しが続行されます。 このトランザクション方針はデフォルトです。
never	TP フレームワークは、このオブジェクトがトランザクション中に呼び出された場合にエラー状態を生成します。
ignore	このオブジェクトが呼び出されたときにトランザクションが活性化されている場合、そのトランザクションはオペレーション呼び出しが完了するまで中断します。このトランザクション方針により、この方針が割り当てられているオブジェクトにトランザクションが伝達されるのを防ぐことができます。

これらの方針をアプリケーションのオブジェクトに割り当てるには、次の手順に従います。

1. 次の例のように、`genicf` コマンドを入力し、アプリケーションの OMG IDL ファイルを入力として指定して、ICF ファイルを生成します。

```
# genicf university.idl
```

このコマンドによって、`university.icf` というファイルが生成されます。

2. ICF ファイルを編集して、アプリケーションのインターフェイスごとに活性化方針を指定します。次の例に、Basic University サンプル・アプリケーション用に生成された ICF ファイルを示します。デフォルトのオブジェクトの活性化方針は `method` で、デフォルトのトランザクション活性化方針は `optional` であることに注意してください。

```
module POA_UniversityB
{
  implementation CourseSynopsisEnumerator_i
  {
    activation_policy ( method );
    transaction_policy ( optional );
    implements ( UniversityB::CourseSynopsisEnumerator );
  };
};

module POA_UniversityB
{
  implementation Registrar_i
  {
    activation_policy ( method );
    transaction_policy ( optional );
    implements ( UniversityB::Registrar );
  };
};

module POA_UniversityB
{
  implementation RegistrarFactory_i
  {
    activation_policy ( method );
    transaction_policy ( optional );
    implements ( UniversityB::RegistrarFactory );
  };
};
```

3. 生成するスケルトン・ファイルとインプリメンテーション・ファイルのインターフェイスの数を制限する場合、それらのインターフェイスをインプリメントするインプリメンテーション・ブロックを ICF ファイルから削除できます。前掲の ICF コードの例では、`RegistrarFactory` インターフェイスのスケルトン・ファイルとインプリメンテーション・ファイルが生成されないようにするには、次の行を削除します。

```
implementation RegistrarFactory_i
{
  activation_policy ( method );
  transaction_policy ( optional );
};
```

```
implements ( UniversityB::RegistrarFactory );  
};
```

4. ICF ファイルを IDL コンパイラに受け渡して、指定された方針に対応するスケルトン・ファイルとインプリメンテーション・ファイルを生成します。詳細については、ページ 2-5 の「スケルトンとインプリメンテーション・ファイルの生成」を参照してください。

ステップ 5: サーバ・アプリケーションのコンパイルとリンク

Server オブジェクトとオブジェクト・インプリメンテーションのコードの記述が済んだら、サーバ・アプリケーションをコンパイルおよびリンクします。

CORBA サーバ・アプリケーションをコンパイルおよびリンクするには、`buildobjserver` コマンドを使用します。`buildobjserver` コマンドの形式は次のとおりです。

```
buildobjserver [-o servername] [options]
```

`buildobjserver` コマンドの構文要素を次に説明します。

- `-o servername` は、このコマンドで生成されるサーバ・アプリケーションの名前を表します。
- `options` は、`buildobjserver` コマンドのコマンド行オプションを表します。

University サンプル・アプリケーションのコンパイルとリンクの詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。`buildobjserver` コマンドの詳細については、『BEA Tuxedo コマンド・リファレンス』を参照してください。

マルチスレッド CORBA サーバ・アプリケーションの設計とビルドには、特別な考慮事項が存在します。詳細については、ページ 4-13 の「`buildobjserver` コマンドの使い方」を参照してください。相互参照が当たっていません (2003/02/27)

注記 IBM AIX 4.3.3 システム上で BEA Tuxedo ソフトウェアを実行する場合、`-brt1` コンパイラ・オプションを使用して CORBA アプリケーションを再コンパイルする必要があります。

ステップ 6: サーバ・アプリケーションのデプロイ

システム管理者は、この節で説明する手順を使用して CORBA サーバ・アプリケーションをデプロイします。University サンプル・アプリケーションのビルドとデプロイの詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。

サーバ・アプリケーションをデプロイするには、以下の手順に従います。

1. サーバ・アプリケーションの実行可能ファイルを、目的の BEA Tuxedo ドメインの一部であるマシンの適切なディレクトリに格納します。
2. 一般のテキスト・エディタを使用して、アプリケーションのコンフィギュレーション・ファイル (`UBBCONFIG` ファイル) を作成します。
3. CORBA サーバ・アプリケーションを起動するマシンで、次の環境変数を設定します。
 - `TUXCONFIG`。この変数は、`UBBCONFIG` ファイルの `TUXCONFIG` エントリと正確に一致する必要があります。この変数は、アプリケーションの `UBBCONFIG` ファイルの格納場所またはパスを表します。
 - `APPDIR`。この変数は、アプリの実行可能ファイルが存在するディレクトリを表します。

4. BEA Tuxedo ドメインで稼働しているすべてのマシン、または BEA Tuxedo ドメインに接続されているすべてのマシンで、`TUXDIR` 環境変数を設定します。この環境変数は、BEA Tuxedo ソフトウェアがインストールされている場所を指し示します。
5. 次のコマンドを入力して、`TUXCONFIG` ファイルを作成します。

```
tmloadcf -y application-ubbconfig-file
```

コマンド行引数の `application-ubbconfig-file` は、アプリケーションの `UBBCONFIG` ファイルの名前を表します。古い `TUXCONFIG` ファイルを削除してからこのコマンドを実行しなければならない場合があることに注意してください。

6. 次のコマンドを入力して、CORBA サーバ・アプリケーションを起動します。

```
tmboot -y
```

`UBBCONFIG` ファイルをリロードせずにサーバ・アプリケーションを再起動できます。

University サンプル・アプリケーションの詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。CORBA アプリケーション用の `UBBCONFIG` ファイルの作成の詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。

開発とデバッグのヒント

ここでは、以下の内容について説明します。

- CORBA 例外とユーザ・ログの使い方
- コールバック・メソッドのエラー状態の検出
- OMG IDL インターフェイスのバージョンと修正に関する一般的な注意事項
- `Tobj_ServantBase::deactivate_object()` オペレーションでの状態処理に関する注意

CORBA 例外とユーザ・ログの使い方

ここでは、以下の内容について説明します。

- 例外のクライアント・アプリケーション・ビュー
- 例外のサーバ・アプリケーション・ビュー

例外のクライアント・アプリケーション・ビュー

クライアント・アプリケーションが CORBA オブジェクトのオペレーションを呼び出した場合、その呼び出しの結果として例外が返される場合があります。クライアント・アプリケーションに返される有効な例外は次のとおりです。

- すべての CORBA 準拠の ORB によって認識される標準の CORBA 定義例外
- OMG IDL で定義され、クライアント・アプリケーションがそのスタブまたはインターフェイス・リポジトリを介して認識する例外

BEA Tuxedo システムは、これらの CORBA 定義の制限に違反することがないように動作します。詳細については、ページ 2-20 の「例外のサーバ・アプリケーション・ビュー」で説明します。相互参照が当たっていません (2003/02/27)

クライアント・アプリケーションが認識する例外セットは制限されているので、クライアント・アプリケーションは原因が不明な例外をキャッチする場合があります。BEA Tuxedo システムは、こうした例外を可能な限りユーザ・ログの説明メッセージで補足します。このメッセージは、エラー状態の検出とデバッグに役立ちます。これらのケースについては、次の節で説明します。

例外のサーバ・アプリケーション・ビュー

ここでは、以下の内容について説明します。

- BEA Tuxedo システムによって生成され、アプリケーション・コードによってキャッチされる例外

- CORBA オブジェクトのオペレーションの呼び出し中にアプリケーション・コードによって生成された例外を BEA Tuxedo システムが処理する方法

BEA Tuxedo システムによって生成され、アプリケーション・コードによってキャッチされる例外

BEA Tuxedo システムは、TP オブジェクトのオペレーションが呼び出された場合、次の例外をアプリケーションに返す場合があります。

- CORBA 定義のシステム例外
- TobjS_c.h ファイルに定義される CORBA UserExceptions。このファイルに定義される例外の OMG IDL は次のとおりです。

```
interface TobjS {
    exception AlreadyRegistered { };
    exception ActivateObjectFailed { string reason; };
    exception ApplicationProblem { };
    exception CannotProceed { };
    exception CreateServantFailed { string reason; };
    exception DeactivateObjectFailed { string reason; };
    exception IllegalInterface { };
    exception IllegalOperation { };
    exception InitializeFailed { string reason; };
    exception InvalidDomain { };
    exception InvalidInterface { };
    exception InvalidName { };
    exception InvalidObject { };
    exception InvalidObjectId { };
    exception InvalidServant { };
    exception NilObject { string reason; };
    exception NoSuchElement { };
    exception NotFound { };
    exception OrbProblem { };
    exception OutOfMemory { };
    exception Overflow { };
    exception RegistrarNotAvailable { };
    exception ReleaseFailed { string reason; };
    exception TpfProblem { };
    exception UnknownInterface { };
}
```

CORBA オブジェクトのオペレーションの呼び出し中にアプリケーション・コードによって生成された例外を BEA Tuxedo システムが処理する方法

サーバ・アプリケーションは、クライアント呼び出し中に次の場所で例外を生成する場合があります。

- `Server::create_servant`、`Tobj_ServantBase::activate_object()`、および `Tobj_ServantBase::deactivate_object()` コールバック・メソッド
- 呼び出されたオペレーションのインプリメンテーション・コード

サーバ・アプリケーションは次のタイプの例外を生成する可能性があります。

- CORBA 定義のシステム例外
- OMG IDL に定義される CORBA ユーザ定義例外
- `TobjS_c.h` ファイルに定義される CORBA ユーザ定義例外。次の例外は、BEA Tuxedo システムがユーザ・ログにメッセージを送るためにサーバ・アプリケーションで使用されます。このメッセージは、トラブルシューティングに役立ちます。

```
interface TobjS {  
    exception ActivateObjectFailed { string reason; };  
    exception CreateServantFailed { string reason; };  
    exception DeactivateObjectFailed { string reason; };  
    exception InitializeFailed { string reason; };  
    exception ReleaseFailed { string reason; };  
}
```

- その他の C++ 例外型

サーバ・アプリケーション・コードによって生成され、サーバ・アプリケーションによってキャッチされないすべての例外は、BEA Tuxedo システムによってキャッチされます。これらの例外がキャッチされると、次のいずれかの処理が発生します。

- 例外は変換されずにクライアント・アプリケーションに返されます。
- 例外は標準 CORBA 例外に変換され、クライアント・アプリケーションに返されます。
- 例外は標準 CORBA 例外に変換され、次のアクションが発生します。

- 例外がクライアント・アプリケーションに返されます。
- エラーに関する説明情報を含む 1 つまたは複数のメッセージがユーザ・ログに送られます。この説明情報は、サーバ・アプリケーション・コードか、または BEA Tuxedo システムから生成されます。

以下の節では、CORBA オブジェクトに対するクライアント呼び出し中にサーバ・アプリケーションによって生成される例外を BEA Tuxedo システムがどのように処理するかについて説明します。

`Server::create_servant()` オペレーションで生成された例外

例外が `Server::create_servant()` オペレーションで生成された場合、次の処理が行われます。

- `CORBA::OBJECT_NOT_EXIST` 例外がクライアント・アプリケーションに返されます。
- 生成された例外が `TobjS::CreateServantFailed` の場合、メッセージがユーザ・ログに送られます。例外のコンストラクタに `reason` 文字列が指定されている場合、その `reason` 文字列もメッセージの一部として書き込まれます。
- `Tobj_ServantBase::activate_object()` オペレーションと `Tobj_ServantBase::deactivate_object()` オペレーションは呼び出されません。クライアントによって要求されたオペレーションは呼び出されません。

`Tobj_ServantBase::activate_object()` オペレーションで生成された例外

例外が `Tobj_ServantBase::activate_object()` オペレーションで生成された場合、次の処理が行われます。

- `CORBA::OBJECT_NOT_EXIST` 例外がクライアント・アプリケーションに返されます。
- 生成された例外が `TobjS::ActivateObjectFailed` の場合、メッセージがユーザ・ログに送られます。例外のコンストラクタに `reason` 文字列が指定されている場合、その `reason` 文字列もメッセージの一部として書き込まれます。

- クライアントによって要求されたオペレーションと
`Tobj_ServantBase::deactivate_object()` オペレーションは呼び出されません。

オペレーション・インプリメンテーションで生成された例外

BEA Tuxedo システムでは、オペレーション・インプリメンテーションは、CORBA システム例外、またはクライアント・アプリケーションによって認識され、OMG IDL に定義されるユーザ定義例外のいずれかをスローする必要があります。これらのタイプの例外がオペレーション・インプリメンテーションによってスローされた場合、BEA Tuxedo システムは次のいずれかの状態が存在しない限り、それらをクライアント・アプリケーションに戻します。

- オブジェクトに `always` トランザクション方針が設定されており、BEA Tuxedo システムがオブジェクトの呼び出し時に自動的にトランザクションを開始した場合。この場合、トランザクションは BEA Tuxedo システムによって自動的にロールバックされます。クライアント・アプリケーションはこのトランザクションを認識しないので、BEA Tuxedo システムは、クライアントがトランザクションを開始した場合に生成される `CORBA:: TRANSACTION_ROLLEDBACK` 例外ではなく `CORBA::OBJ_ADAPTER` CORBA システム例外を生成します。
- 例外が `TobjS_c.h` ファイルに定義されている場合。この場合、例外は `CORBA::BAD_OPERATION` 例外に変換され、クライアント・アプリケーションに戻されます。また、次のメッセージがユーザ・ログに送られます。

```
"WARN: Application didn't catch TobjS exception. TP Framework  
throwing CORBA::BAD_OPERATION."
```

例外が `TobjS::IllegalOperation` の場合、次の補足メッセージが書き込まれ、アプリケーションにコーディング・エラーが存在する可能性があることを開発者に警告します。

```
"WARN: Application called TP::deactivateEnable() illegally and  
didn't catch TobjS exception."
```

これは、`TP::deactivateEnable()` オペレーションが、`transaction` 活性化方針が割り当てられているオブジェクトの内部で呼び出された場合に発生する可能性があります。アプリケーション制御の非活性化はトランザクション・バウンド・オブジェクトではサポートされません。

- BEA Tuxedo システムが、クライアント呼び出しに続いて内部システム例外を生成した場合。この場合、CORBA::INTERNAL 例外がクライアントに返されます。一般にこれは、オブジェクトが活性化されているプロセスに関する重大なシステム問題を示します。

CORBA 仕様で定義されているとおり、クライアントに送り返される応答には、オペレーション・インプリメンテーションからの結果値か、オペレーション・インプリメンテーションでスローされた例外のいずれかが含まれ、両方が含まれることはありません。最初のケース、つまり、応答ステータス値が NO_EXCEPTION - の場合、応答にはオペレーションの戻り値と任意の inout または out 引数値が含まれます。それ以外のケース、つまり、応答ステータス値が USER_EXCEPTION または SYSTEM_EXCEPTION の場合、応答には例外の符号化が含まれます。

Tobj_ServantBase::deactivate_object() オペレーションで生成された例外

例外が Tobj_ServantBase::deactivate_object() オペレーションで生成された場合、次の処理が発生します。

- 例外はクライアント・アプリケーションに返されません。
- 生成された例外が TobjS::DectivateObjectFailed の場合、メッセージがユーザ・ログに送られます。例外のコンストラクタに reason 文字列が指定されている場合、その reason 文字列もメッセージの一部として書き込まれます。
- TobjS::DeactivateObjectFailed 以外の例外のメッセージがユーザ・ログに送られます。このメッセージは、BEA Tuxedo システムによってキャッチされた例外の型を示します。

オブジェクト・インスタンスの受け渡し時に生成された CORBA マーシャリング例外

ORB は、オブジェクト・インスタンスをオブジェクト・リファレンスとしてマーシャリングできません。たとえば、次のコードでファクトリ・リファレンスを受け渡すと、BEA Tuxedo システムで CORBA マーシャリング例外が発生します。

```
connection::setFactory(this);
```

オブジェクト・インスタンスを受け渡すには、次の例のように、プロキシ・オブジェクト・リファレンスを作成して、そのプロキシを代わりに受け渡します。

```
CORBA::Object myRef = TP::get_object_reference();
ResultSetFactory factoryRef = ResultSetFactoryHelper::_narrow(myRef);
connection::setFactoryRef(factoryRef);
```

Detecting Error Conditions in the Callback Methods

BEA Tuxedo システムには、メッセージ文字列を指定できる定義済みの例外セットが用意されています。TP フレームワークは、アプリケーション・コードが次のコールバック・メソッドでエラーを取得した場合、これらのメッセージ文字列をユーザ・ログに書き込みます。

- Tobj_ServantBase::activate_object()
- Tobj_ServantBase::deactivate_object()
- Server::create_servant()
- Server::initialize()
- Server::release()

これらの例外は、例外の発生原因に関する明確な情報を送信するための便利なデバッグ支援機能として使用できます。TP フレームワークは、これらのメッセージをユーザ・ログにのみ書き込みます。これらのメッセージは、クライアント・アプリケーションには返されません。

これらのメッセージは、次の例外で指定します。これらの例外では、オプションで reason 文字列を指定できます。

例外	この例外を生成するコールバック・メソッド
ActivateObjectFailed	Tobj_ServantBase::activate_object()
DeactivateObjectFailed	Tobj_ServantBase::deactivate_object()
CreateServantFailed	Server::create_servant()

例外	この例外を生成するコールバック・メソッド
InitializeFailed	Server::initialize()
ReleaseFailed	Server::release()

メッセージ文字列をユーザ・ログに送るには、次の例のように、その文字列を例外に指定します。

```
throw CreateServantFailed("Unknown interface");
```

これらの例外をスローする場合、reason 文字列パラメータが指定されている必要があります。これらの例外の 1 つで reason 文字列を指定しない場合は、次の例のように、二重引用符を入力する必要があります。

```
throw ActivateObjectFailed("");
```

OMG IDL インターフェイスのバージョンと修正に関する一般的な注意事項

Server オブジェクトの `Server::create_servant()` オペレーションのインプリメンテーションは、そのインターフェイス ID に基づいてオブジェクトをインスタンス化します。このインターフェイス ID は、ファクトリが `TP::create_object_reference()` オペレーションを呼び出したときにファクトリに指定されるインターフェイス ID と同じである必要があります。インターフェイス ID が一致しない場合、通常 `Server::create_servant()` オペレーションで例外が発生するか、または NULL サーバントが返されます。この場合、BEA Tuxedo システムは、`CORBA::OBJECT_NOT_EXIST` 例外をクライアント・アプリケーションに返します。BEA Tuxedo システムは、`TP::create_object_reference()` オペレーションでインターフェイス ID の検証を行いません。

このような状態は、開発の過程で、インターフェイスの異なるバージョンが開発されるか、IDL ファイルに対して多くの変更が行われる場合に発生する可能性があります。OMG IDL にインターフェイス ID の文字列定数を指定し、これらの定数をファクトリと `Server::create_servant()` オペレーショ

ンで使用する場合でも、オブジェクト・インプリメンテーションとファクトリが異なる実行可能ファイルに存在する場合は、不一致が発生する可能性があります。多くの場合、この問題の診断は困難です。

こうした問題を避けるには、開発中に次の予防的プログラミング手法を検討する必要があります。このコードは、アプリケーションのデバッグ・バージョンにのみ記述する必要があります。実働バージョンでは受け入れられない性能の低下が発生する可能性があるからです。

- ファクトリが `TP::create_object_reference()` オペレーションを呼び出す直前に、インターフェイス・リポジトリをチェックして必要なインターフェイスが存在するかどうかを調べるコードを組み込みます。すべてのアプリケーション OMG IDL が最新のものになっており、インターフェイス・リポジトリにロードされていることを確認します。このチェックでインターフェイス ID が見つからない場合、不一致が存在すると見なすことができます。
- ファクトリでの `TP::create_object_reference()` オペレーションの呼び出しの後に、オブジェクトを ping するコード、つまり、オブジェクトの任意のオペレーション（通常何も行わないオペレーション）を呼び出すコードを組み込みます。この呼び出しが `CORBA::OBJECT_NOT_EXIST` 例外を生成する場合、インターフェイス ID の不一致が存在します。オブジェクトを ping すると、そのオブジェクトが活性化され、その活性化に関連するオーバーヘッドが発生することに注意してください。

Tobj_ServantBase::deactivate_object() での状態処理に関する注意

`Tobj_ServantBase::deactivate_object()` オペレーションは、オブジェクトの活性化境界に達したときに呼び出されます。このオペレーションのインプリメンテーションでは、オブションで永続状態をディスクに書き込むことができます。このオペレーションで生成された例外はクライアント・アプリケーションに返されないことを理解しておくことが重要です。クライアント・アプリケーションは、オブジェクトがトランザクションに参加していない限り、このオペレーションで生成されたエラー状態を認識しません。このため、このオペレーションで状態が正常に書き込まれたかどうかを知ることが重要である場合は、トランザクションを使用することをお勧めします。

`Tobj_ServantBase::deactivate_object()` オペレーションで状態を書き込むことを選択し、クライアント・アプリケーションがその書き込みオペレーションの結果を知る必要がある場合、次のことを行うことをお勧めします。

- オブジェクトの状態に影響を及ぼす各オペレーションがトランザクションの中で呼び出され、非活性化がトランザクション境界の中で発生するようにします。そのためには、`method` 活性化方針か `transaction` 活性化方針のいずれかを使用します。また、`TP::deactivateEnable()` オペレーションがトランザクション境界の内部で呼び出される場合は、`process` 活性化方針を使用することもできます。
- オブジェクト状態の書き込み中にエラーが発生した場合、`COSTransactions::Current::rollback_only()` オペレーションを呼び出してトランザクションがロールバックされるようにします。これにより、クライアント・アプリケーションは次の例外を確実に受け取ることができます。
 - クライアント・アプリケーションがトランザクションを開始した場合、クライアント・アプリケーションは `CORBA::TRANSACTION_ROLLEDBACK` 例外を受け取ります。
 - BEA Tuxedo システムがトランザクションを開始した場合、クライアント・アプリケーションは `CORBA::OBJ_ADAPTER` 例外を受け取ります。

トランザクションを使用しない場合は、

`Tobj_ServantBase::deactivate_object()` オペレーションを使用せずに、オブジェクトの個々のオペレーションのスコープの中でオブジェクト状態を書き込むことをお勧めします。エラーが発生した場合、オペレーションはクライアント・アプリケーションに返される例外を生成できます。

サーバント・プール

ページ 1-21 の「サーバント・プールと状態を持たないオブジェクト」で説明したように、サーバント・プールを使用すると、メソッド・バウンド・オブジェクトとトランザクション・バウンド・オブジェクトのオブジェクト・インスタンス化のコストを削減できます。

サーバント・プールの機能

通常、オブジェクトの非活性化中 (TP フレームワークが

`Tobj_ServantBase::deactivate_object()` オペレーションを呼び出したとき) に、TP フレームワークはオブジェクトのサーバントを削除します。ただし、サーバント・プールを使用する場合、TP フレームワークはオブジェクトの非活性化時にサーバントを削除しません。代わりに、サーバ・アプリケーションはプール内のサーバントへのポインタを保持します。それ以降、そのプール内のサーバントによって処理可能な要求がクライアントから送られてくると、サーバ・アプリケーションはそのサーバントを再利用して新しいオブジェクト ID を割り当てます。サーバントがプールから再利用される場合、TP フレームワークは新しいサーバントを作成しません。

サーバント・プールのインプリメント方法

サーバント・プールは、以下の手順でインプリメントします。

1. Server オブジェクトの `Server::initialize()` オペレーションに、サーバント・プールをセットアップするコードを記述します。サーバント・プールは 1 つまたは複数のサーバントへのポインタのセットで構成され、サーバント・プールのコードでは特定のクラスのサーバントをプールでどのくらい保持するのかを指定します。
2. プールされたサーバントの `Tobj_ServantBase::deactivate_object()` オペレーションに、`TP::application_responsibility()` オペレーションをインプリメントします。`TP::application_responsibility()` オペレーションのインプリメンテーションには、TP フレームワークが `Tobj_ServantBase::deactivate_object()` オペレーションを呼び出したときにサーバント・プールにサーバントへのポインタを格納するコードを記述します。
3. Server オブジェクトの `Server::create_servant()` オペレーションのインプリメンテーションに、クライアント要求が到着したときに次のことを行うコードを記述します。
 - a. プールをチェックして、その要求を満たすことができるサーバントが存在するかどうかを調べる。

- b. サーバントが存在しない場合は、サーバントを作成してその `Tobj_ServantBase::activate_object()` オペレーションを呼び出す。
- c. サーバントが存在する場合は、その `Tobj_ServantBase::activate_object()` オペレーションを呼び出して、クライアント要求に含まれているオブジェクト ID を割り当てる。

注記 このリリースでは、`TP::application_responsibility()` オペレーションのサポートが変更されています。詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

デレゲーション・ベースのインターフェイス・インプリメンテーション

BEA Tuxedo CORBA アプリケーションでオブジェクトをインプリメントする主要な方法は、継承とデレゲーションの 2 つです。オブジェクトが POA スケルトン・クラスから継承され、それによって CORBA オブジェクトとなった場合、そのオブジェクトは継承によってインプリメントされたと言えます。

ただし、POA スケルトン・クラスからの継承が困難または不可能な C++ オブジェクトを CORBA アプリケーションで使用したい場合もあります。たとえば、POA スケルトン・クラスから継承するために大幅な書き換えが必要な C++ オブジェクトなどです。こうした非 CORBA オブジェクトを CORBA アプリケーションで使用するには、そのオブジェクト用の tie クラスを作成します。tie クラスは、POA スケルトン・クラスから継承されます。また、tie クラスには 1 つまたは複数のオペレーションが含まれ、それらはインプリメンテーションのためにレガシー・クラスに委譲されます。これにより、レガシー・クラスはデレゲーションによって CORBA アプリケーションにインプリメントされます。

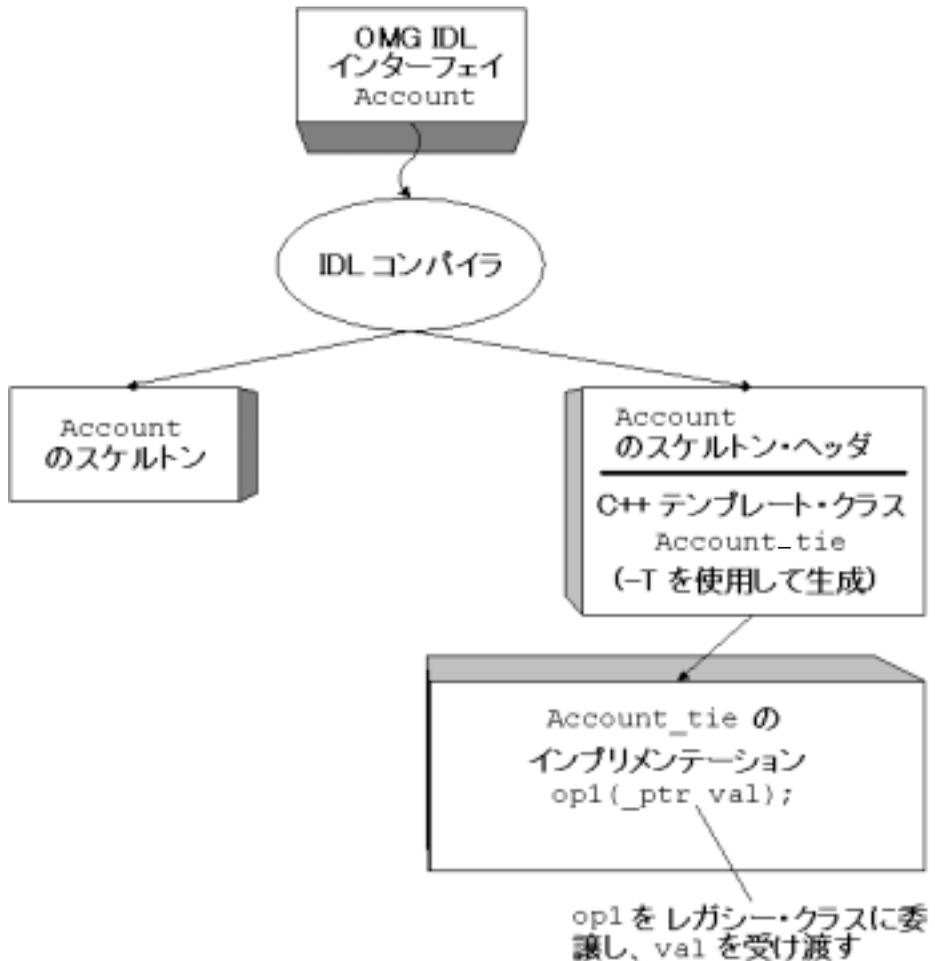
BEA Tuxedo システムの tie クラスについて

デレゲーション・ベースのインターフェイス・インプリメンテーションを作成するには、IDL コンパイラの `-T` コマンド行オプションを使用して、OMG IDL ファイルに定義されている各インターフェイスに対する tie クラス・テンプレートを生成します。

CORBA アプリケーションで tie クラスを使用する場合、Server オブジェクトに `Server::create_servant()` オペレーションをインプリメントする方法も変わります。以降の節では、BEA Tuxedo 製品で tie クラスを使用する方法についてさらに詳しく説明し、`Server::create_servant()` オペレーションをインプリメントしてこれらのクラスをインスタンス化する方法についても説明します。

BEA Tuxedo CORBA では、tie クラスはサーバントであり、したがって基本的にレガシー・クラスのラッパー・オブジェクトとして機能します。

次の図に、レガシー・オブジェクトのラッパーとして機能する、Account インターフェイスの継承の特性を示します。レガシー・オブジェクトには、オペレーション `op1` のインプリメンテーションが含まれています。tie クラスは、`op1` をレガシー・クラスに委譲します。



tie クラスは、クライアント・アプリケーションからは見えません。クライアント・アプリケーションには、tie クラスは自身が呼び出すオブジェクトの完全なインプリメンテーションのように見えます。tie クラスは、ユーザが提供するレガシー・クラスにすべてのオペレーションを委譲します。さらに、tie クラスには次のものが含まれます。

- コンストラクタとデストラクタのコード。このコードは、tie クラスとレガシー・クラスの起動およびシャットダウン手順を処理します。
- ハウスキーピング・コード。このコードは、アクセサなどのオペレーションをインプリメントします。

tie クラスを使用する条件

tie クラスは BEA Tuxedo CORBA に固有のものではなく、CORBA アプリケーションでデレゲーションをインプリメントするための唯一の方法でもありません。ただし、BEA Tuxedo CORBA の tie クラス用の便利な機能を使用すると、それらの tie クラスの基本的なコンストラクタ、デストラクタ、およびハウスキーピング・オペレーションに必要なコーディングの量を大幅に減らすことができます。

tie クラスは、次のいずれかの状況でを使用することをお勧めします。

- POA スケルトン・クラスからの継承が困難または不可能なオブジェクトを CORBA アプリケーションにインプリメントする場合
- レガシー・クラスのインスタンスのすべての呼び出しを 1 つのサーバントから実現できる場合
- CORBA アプリケーションでレガシー・クラスを使用しており、そのインスタンスの存続期間をサーバント・クラスに結びつける場合
- 特定のサーバントの唯一の目的がデレゲーションであり、このため、そのサーバントのほぼすべてのコードがレガシー・オブジェクトの起動、シャットダウン、およびデレゲーションに使用される場合

次の場合、tie クラスの使用はお勧めしません。

- オブジェクト・インスタンスのオペレーションが複数のレガシー・オブジェクトのインスタンスに委譲される場合
- デレゲーションがオブジェクトの目的の一部に過ぎない場合

CORBA アプリケーションに tie クラスを作成する方法

BEA Tuxedo ドメインのアプリケーションに tie クラスを作成するには、次の手順に従います。

1. アプリケーションのオブジェクトと同じように、OMG IDL ファイルに tie クラスのインターフェイス定義を作成します。
2. `-T` オプションを使用して、OMG IDL ファイルをコンパイルします。

IDL コンパイラは、C++ テンプレート・クラスを生成します。このクラスはスケルトンの名前を取り、その後ろに `_tie` という文字列が付加されます。IDL コンパイラは、このテンプレート・クラスをスケルトン・ヘッダ・ファイルに追加します。

IDL コンパイラは、tie クラスのインプリメンテーション・ファイルを生成しないことに注意してください。このファイルは、次のステップで説明するように手動で作成する必要があります。

3. tie クラスのインプリメンテーション・ファイルを作成します。インプリメンテーション・ファイルには、そのオペレーションをレガシー・クラスに委譲するコードを記述します。
4. Server オブジェクトの `Server::create_servant()` オペレーションに、レガシー・オブジェクトをインスタンス化するコードを記述します。

次の例では、tie クラス `POA_Account_tie` のサーバントが作成され、レガシー・クラス `LegacyAccount` がインスタンス化されます。

```
Account * Account_ptr = new LegacyAccount();
AccountFactoryServant = new POA_Account_tie<LegacyAccount> (Account_ptr)
```

注記 UNIX 用の Compaq C++ Tru64 コンパイラで tie クラスをコンパイルする場合、`buildobjserver` コマンドで使用される `CFLAGS` または `CPPFLAGS` 環境変数の定義に `-noimplicit_include` オプションを指定する必要があります。このオプションを指定すると、C++ コンパイラはサーバ・スケルトン・ファイル (`_s.h`) がインクルードされる場所にサーバ・スケルトン定義ファイル (`_s.cpp`) を自動的に組み込み

ません。これは、複数定義のシンボル・エラーを回避するために必要です。Tru64 C++ で tie クラスなどのクラス・テンプレートを使用する方法については、Compaq の出版物を参照してください。

2 BEA Tuxedo CORBA サーバ・アプリケーションの作成手順

3 Basic CORBA サーバ・アプリケーションの設計とインプリメント

この章では、Basic University サンプル・アプリケーションを例として使用して、CORBA サーバ・アプリケーションを設計およびインプリメントする方法について説明します。この章の内容は、インプリメントするアプリケーションの設計が完了し、OMG IDL に定義されていることを前提にしています。この章では、サーバ・アプリケーション向けの設計およびインプリメンテーションの選択肢に焦点を当てます。

ここでは、以下の内容について説明します。

- Basic University サンプル・アプリケーションのしくみ。このトピックでは、設計とインプリメンテーションの考慮事項について説明します。
- University サーバ・アプリケーションの設計上の考慮事項。このトピックでは、次のトピックについて包括的に説明します。
 - オブジェクト・リファレンスの生成に関する設計上の考慮事項
 - オブジェクト状態の管理に関する設計上の考慮事項
 - 永続状態情報の処理に関する設計上の考慮事項

- Basic サンプル・アプリケーションがデザイン・パターンを適用する方法
- BEA Tuxedo システムに組み込まれた性能効率化
- 状態を持つオブジェクトの事前活性化

Basic University サンプル・アプリケーションのしくみ

Basic University サンプル・アプリケーションを使用すると、学生は中央の University データベースからコース情報を参照できます。この Basic サンプル・アプリケーションでは、学生は次のことを行うことができます。

- 検索文字列を指定することで、データベースからコース概要を参照します。サーバ・アプリケーションは、検索文字列に含まれているタイトル、教授、または説明に一致するすべてのコースの概要を返します。クライアント・アプリケーションに返されるコース概要には、コースの番号とタイトルしか含まれません。
- 特定のコースに関する詳細情報を参照します。指定したコースの詳細情報には、コース概要のほかに次の情報が含まれます。
 - 受講料
 - 履修単位
 - クラス・スケジュール
 - 座席数
 - 登録学生数
 - 教授
 - 説明

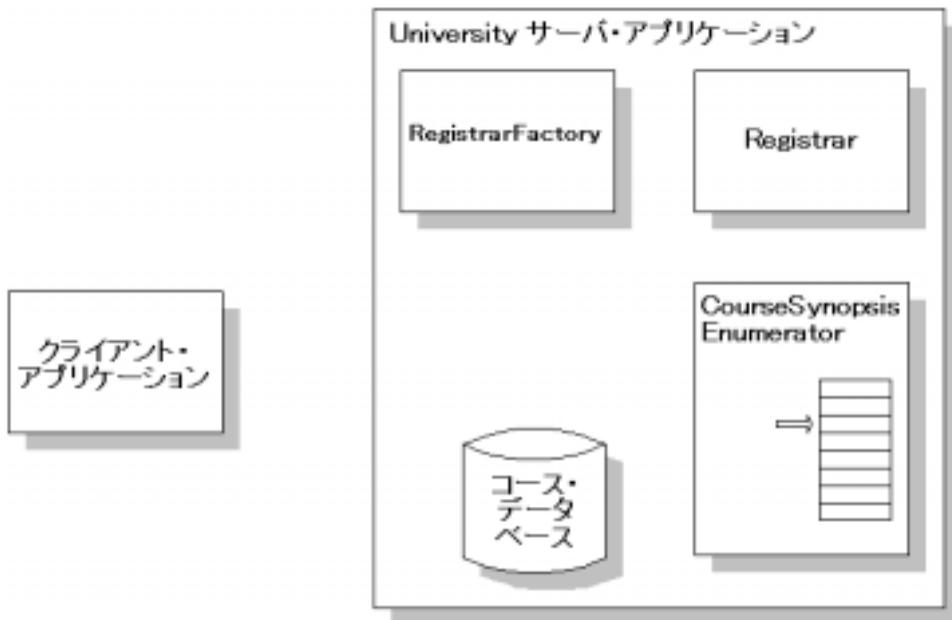
Basic University サンプル・アプリケーションの OMG IDL

Basic University サンプル・アプリケーションの OMG IDL ファイルには、次のインターフェイスが定義されます。

インターフェイス	説明	オペレーション
RegistrarFactory	Registrar オブジェクトのオブジェクト・リファレンスを作成します。	find_registrar()
Registrar	データベースからコース情報を取得します。	get_courses_synopsis() get_courses_details()
CourseSynopsisEnumerator	検索基準に一致するコースの概要をデータベースからフェッチして、それらをメモリに読み込みます。	get_next_n() destroy()

図 3-1 に、Basic University サンプル・アプリケーションを示します。

図 3-1 Basic University サンプル・アプリケーション



Basic University サンプル・アプリケーションで何が発生するかを説明するために、次のイベント・グループについて説明します。

- アプリケーションの起動 — サーバ・アプリケーションが起動し、クライアント・アプリケーションが `Registrar` オブジェクトのオブジェクト・リファレンスを取得するときです。
- コース概要の参照 — クライアント・アプリケーションが要求を送信してコース概要を参照するときです。
- コース詳細の参照 — クライアント・アプリケーションが要求を送信して特定のコース・リストの詳細を参照するときです。

アプリケーションの起動

次に、Basic クライアント・アプリケーションおよびサーバ・アプリケーションが起動し、クライアント・アプリケーションが Registrar オブジェクトのオブジェクト・リファレンスを取得するときに行われる一般的な一連のイベントを示します。

1. Basic クライアント・アプリケーションおよびサーバ・アプリケーションが起動し、クライアント・アプリケーションは FactoryFinder から RegistrarFactory オブジェクトのオブジェクト・リファレンスを取得します。
2. RegistrarFactory オブジェクトのリファレンスを使用して、クライアント・アプリケーションは RegistrarFactory オブジェクトの `find_registrar()` オペレーションを呼び出します。
3. RegistrarFactory はメモリ内に存在しません (このオブジェクトに対する要求が以前にサーバ・プロセスに到着していないため)。このため、TP フレームワークは Server オブジェクトの `Server::create_servant()` オペレーションを呼び出して RegistrarFactory オブジェクトをインスタンス化します。
4. インスタンス化が済んだら、RegistrarFactory オブジェクトの `find_registrar()` オペレーションが呼び出されます。RegistrarFactory オブジェクトは、Registrar のオブジェクト・リファレンスを作成してクライアント・アプリケーションに返します。

コース概要の参照

次に、学生がコース概要のリストを参照するときに発生する一連のイベントを順番に示します。

1. Registrar オブジェクトのオブジェクト・リファレンスを使用して、クライアント・アプリケーションは `get_courses_synopsis()` オペレーションを呼び出し、次のものを指定します。
 - データベースからコース概要を検索するための検索文字列

3 Basic CORBA サーバ・アプリケーションの設計とインプリメント

- 変数 `number_to_get` で表され、返される概要リストのサイズを指定する整数
2. Registrar はメモリ内に存在しません (このオブジェクトに対する要求が以前にサーバ・プロセスに到着していないため)。このため、TP フレームワークは Server オブジェクトにインプリメントされている `Server::create_servant()` オペレーションを呼び出します。これにより、Registrar オブジェクトがサーバ・マシンのメモリ内にインスタンス化されます。
 3. Registrar オブジェクトはクライアント要求を受信し、`CourseSynopsisEnumerator` オブジェクトのオブジェクト・リファレンスを作成します。`CourseSynopsisEnumerator` オブジェクトは、データベースからコース概要をフェッチするために Registrar オブジェクトによって呼び出されます。

`CourseSynopsisEnumerator` オブジェクトのオブジェクト・リファレンスを作成するために、Registrar オブジェクトは次のことを行います。

 - a. `CourseSynopsisEnumerator` オブジェクトの一意的 ID を生成します。
 - b. `CourseSynopsisEnumerator` オブジェクトのオブジェクト ID を生成します。この ID は、前のステップで生成された一意の ID とクライアントによって指定された検索文字列を連結したものです。
 - c. `CourseSynopsisEnumerator` オブジェクトのインターフェイス・リポジトリ ID をそのインターフェイス・タイプ・コードから取得します。
 - d. `TP::create_object_reference()` オペレーションを呼び出します。このオペレーションにより、最初のクライアント要求で必要な `CourseSynopsisEnumerator` オブジェクトのオブジェクト・リファレンスが作成されます。
 4. 前のステップで作成したオブジェクト・リファレンスを使用して、Registrar オブジェクトは `CourseSynopsisEnumerator` オブジェクトの `get_next_n()` オペレーションを呼び出して、リスト・サイズを受け渡します。ステップ 1 で説明したとおり、このリスト・サイズはパラメータ `number_to_get` によって表されます。

5. TP フレームワークは、Server オブジェクトの `Server::create_servant()` オペレーションを呼び出して、`CourseSynopsisEnumerator` オブジェクトをインスタンス化します。
6. TP フレームワークは、`CourseSynopsisEnumerator` オブジェクトの `activate_object()` オペレーションを呼び出します。このオペレーションは、次の2つのことを行います。
 - その OID から検索基準を抽出します。
 - その検索基準を使用して、一致するコース概要をデータベースからフェッチし、それらをサーバ・マシンのメモリに読み込みます。
7. `CourseSynopsisEnumerator` オブジェクトは、`Registrar` オブジェクトに次の情報を返します。
 - 戻り値の `CourseSynopsisList` に指定されるコース概要リスト。これは、最初のコース概要リストを含む `sequence` です。
 - 検索基準に一致するが、まだ返されていないコース概要の数。これは、`number_remaining` パラメータによって指定されます。
8. `Registrar` オブジェクトは、クライアント・アプリケーションに `CourseSynopsisEnumerator` オブジェクト・リファレンスを返します。また、そのオブジェクトから取得した次の情報も返します。
 - 最初のコース概要リスト
 - `number_remaining` 変数

`number_remaining` 変数が0の場合、`Registrar` オブジェクトは `CourseSynopsisEnumerator` オブジェクトの `destroy()` オペレーションを呼び出し、クライアント・アプリケーションにニル・リファレンスを返します。
9. クライアント・アプリケーションは、一致する次の概要リストを取得するための要求を直接 `CourseSynopsisEnumerator` オブジェクトに送信します。
10. `CourseSynopsisEnumerator` オブジェクトはそのクライアント要求を満たすと共に、更新した `number_remaining` 変数を返します。

11. クライアント・アプリケーションは、`CourseSynopsisEnumerator` オブジェクトとのやり取りを終了すると、`CourseSynopsisEnumerator` オブジェクトの `destroy()` オペレーションを呼び出します。この結果、`CourseSynopsisEnumerator` オブジェクトは `TP::deactivateEnable()` オペレーションを呼び出します。
12. TP フレームワークは、`CourseSynopsisEnumerator` オブジェクトの `deactivate_object()` オペレーションを呼び出します。この結果、`CourseSynopsisEnumerator` オブジェクトが保持しているコース概要のリストがサーバ・マシンのメモリから削除されます。これにより、`CourseSynopsisEnumerator` オブジェクトのサーバントを別のクライアント要求のために再利用できるようになります。

コース詳細の参照

次に、クライアント・アプリケーションがコース詳細を参照するときに行われる一連のイベントを示します。

1. 学生は、詳細を参照したいコースのコース番号を入力します。
2. クライアント・アプリケーションは、`Registrar` オブジェクトの `get_course_details()` オペレーションを呼び出し、コース番号のリストを受け渡します。
3. `Registrar` オブジェクトは、一致するコース番号をデータベースから検索し、指定された各コースの詳細を含んだリストを返します。このリストは、`CourseDetailsList` 変数に格納されます。これは、コースの詳細を含んだ一連の `struct` です。

University サーバ・アプリケーションの設計上の考慮事項

Basic University サンプル・アプリケーションには、University サーバ・アプリケーションが含まれます。このサーバ・アプリケーションは、CORBA サーバ・アプリケーションの基本的な設計問題を処理します。この節では、以下のトピックについて説明します。

- オブジェクト・リファレンスの生成に関する設計上の考慮事項
- オブジェクト状態の管理に関する設計上の考慮事項
- 永続状態情報の処理に関する設計上の考慮事項
- Basic サンプル・アプリケーションがデザイン・パターンを適用する方法

この節では、さらに次の2つのトピックについても説明します。

- BEA Tuxedo システムに組み込まれた性能効率化
- 状態を持つオブジェクトの事前活性化

オブジェクト・リファレンスの生成に関する設計上の考慮事項

Basic クライアント・アプリケーションでは、University サーバ・アプリケーションによって管理される次のオブジェクトのリファレンスが必要です。

- RegistrarFactory オブジェクト
- Registrar オブジェクト
- CourseSynopsisEnumerator オブジェクト

次の表に、これらのリファレンスがどのように生成されて返されるかを示します。

3 Basic CORBA サーバ・アプリケーションの設計とインプリメント

オブジェクト	オブジェクト・リファレンスが生成されて返されるしくみ
RegistrarFactory	<p>RegistrarFactory オブジェクトのオブジェクト・リファレンスは Server オブジェクトで生成されます。Server オブジェクトは、RegistrarFactory オブジェクトを FactoryFinder に登録します。クライアント・アプリケーションは、RegistrarFactory オブジェクトのリファレンスを FactoryFinder から取得します。</p> <p>Basic University サーバ・アプリケーション・プロセスには、RegistrarFactory オブジェクトは 1 つしか存在しません。</p>
Registrar	<p>Registrar オブジェクトのオブジェクト・リファレンスは RegistrarFactory オブジェクトによって生成され、クライアント・アプリケーションが find_registrar() オペレーションを呼び出したときに返されます。Registrar オブジェクトに対して作成されるオブジェクト・リファレンスは常に同じです。このオブジェクト・リファレンスには一意な OID は含まれません。</p> <p>Basic University サーバ・アプリケーション・プロセスには、Registrar オブジェクトは 1 つしか存在しません。</p>

オブジェクト	オブジェクト・リファレンスが生成されて返されるしくみ
CourseSynopsisEnumerator	<p>CourseSynopsisEnumerator オブジェクトのオブジェクト・リファレンスは、クライアント・アプリケーションが <code>get_courses_synopsis()</code> オペレーションを呼び出したときに Registrar オブジェクトによって作成されます。つまり、Registrar オブジェクトは CourseSynopsisEnumerator オブジェクトのファクトリです。CourseSynopsisEnumerator オブジェクトの設計と使い方については、この章で後述します。</p> <p>Basic University サーバ・アプリケーション・プロセスには、任意の数の CourseSynopsisEnumerator オブジェクトが存在できます。</p>

University サーバ・アプリケーションがオブジェクト・リファレンスを生成する方法について、以下のことに注意してください。

- Server オブジェクトは、RegistrarFactory オブジェクトを FactoryFinder に登録します。この方法により、クライアント・アプリケーションはサーバ・アプリケーション内の基本オブジェクトのオブジェクト・リファレンスを取得するために必要なファクトリを検索できます。
- Registrar オブジェクトのオブジェクト・リファレンスは、RegistrarFactory オブジェクトによって作成されます。これは、クライアント・アプリケーションにオブジェクト・リファレンスを返すためのきわめて一般的かつ基本的な方法を示します。つまり、クライアント・アプリケーションがビジネス・ロジックを実行するために必要な基本オブジェクトのリファレンスを作成して返すための専用ファクトリが存在するということです。
- CourseSynopsisEnumerator オブジェクトのオブジェクト・リファレンスは、登録されたファクトリの外部で作成されます。University サンプル・アプリケーションの場合、これは優れた設計です。これは、CourseSynopsisEnumerator オブジェクトの使い方のためです。つまり、

その存在が特定のクライアント・アプリケーション・オペレーションに固有であるからです。CourseSynopsisEnumerator オブジェクトは、特定のリストと、ほかの問い合わせの結果とは関連性のない結果を返します。

- Registrar はそのオペレーションの 1 つで別のオブジェクトのオブジェクト・リファレンスを作成するので、Registrar オブジェクトはファクトリです。ただし、Registrar オブジェクトはファクトリとして FactoryFinder に登録されません。このため、クライアント・アプリケーションは Registrar オブジェクトのリファレンスを FactoryFinder から取得することはできません。

オブジェクト状態の管理に関する設計上の考慮事項

Basic サンプル・アプリケーションの 3 つのオブジェクトには、それぞれ独自の状態管理の要件が存在します。この節では、それぞれのオブジェクト状態管理の要件について説明します。

RegistrarFactory オブジェクト

RegistrarFactory オブジェクトは、特定のクライアント要求に一意なものである必要はありません。また、このオブジェクトをメモリに保持しておき、クライアント呼び出しごとにこのオブジェクトを活性化および非活性化の負担を回避することは合理的です。したがって、RegistrarFactory オブジェクトには process 活性化方針が割り当てられます。

Registrar オブジェクト

Basic サンプル・アプリケーションは、小規模な環境へのデプロイに適しています。Registrar オブジェクトは、RegistrarFactory オブジェクトに似た特性を備えています。つまり、このオブジェクトは特定のクライアント要求に一意なものである必要がありません。また、呼び出しごとにこのオブジェクトを活性化および非活性化する負担を回避することは合理的です。したがって、Basic サンプル・アプリケーションでは、Registrar オブジェクトには process 活性化方針が割り当てられます。

CourseSynopsisEnumerator オブジェクト

University サーバ・アプリケーションの基本的な設計上の問題は、大きすぎて 1 回の応答ではクライアント・アプリケーションに返すことができないコース概要リストをどのように処理するかです。したがって、この問題の解決策は、次のことが中心となります。

- クライアント・アプリケーションとコース概要を University データベースからフェッチできるオブジェクト間の会話を開始します。
- オブジェクトがクライアント・アプリケーションに最初の概要リストを返します。
- コース概要の残りをメモリに保持しておき、クライアント・アプリケーションが一度にそれらを取得できるようにします。
- 完了時にクライアント・アプリケーションが会話を終了することによって、マシンのリソースを解放します。

University サーバ・アプリケーションは、この解決策をインプリメントする `CourseSynopsisEnumerator` オブジェクトを備えています。このオブジェクトは最初に呼び出されたときに最初の概要リストを返しますが、その後もメモリ内コンテキストに保持されるので、クライアント・アプリケーションは後続の要求で概要の残りを取得できます。メモリ内コンテキストを保持するには、`CourseSynopsisEnumerator` オブジェクトは状態を持たなければなりません。つまり、このオブジェクトは複数のクライアント呼び出しにわたってメモリ内に常駐します。

クライアントが `CourseSynopsisEnumerator` オブジェクトとの会話を終了するときに、このオブジェクトにはメモリからフラッシュされるための手段が必要です。このため、`CourseSynopsisEnumerator` オブジェクトの適切な状態管理の方法は、`process` 活性化方針を割り当て、CORBA アプリケーション制御の非活性化機能をインプリメントすることです。

アプリケーション制御の非活性化は、そのオブジェクトの `destroy()` オペレーションによってインプリメントされます。

次のコード例に、`CourseSynopsisEnumerator` オブジェクトの `destroy()` オペレーションを示します。

```
void CourseSynopsisEnumerator_i::destroy()  
{
```

3 Basic CORBA サーバ・アプリケーションの設計とインプリメント

```
// クライアントが列挙子の「破棄」を呼び出すと、
// このオブジェクトは「消滅」する必要がある。
// そのためには、TP フレームワークに
// このオブジェクトとの会話が完了したことを通知する

TP::deactivateEnable();
}
```

Basic University サンプル・アプリケーションの ICF ファイル

次のコード例に、Basic サンプル・アプリケーション用の ICF ファイルを示します。

```
module POA_UniversityB
{
    implementation CourseSynopsisEnumerator_i
    {
        activation_policy ( process ) ;
        transaction_policy ( optional ) ;
        implements ( UniversityB::CourseSynopsisEnumerator ) ;
    };
    implementation Registrar_i
    {
        activation_policy ( process ) ;
        transaction_policy ( optional ) ;
        implements ( UniversityB::Registrar ) ;
    };
    implementation RegistrarFactory_i
    {
        activation_policy ( process ) ;
        transaction_policy ( optional ) ;
        implements ( UniversityB::RegistrarFactory ) ;
    };
};
```

永続状態情報の処理に関する設計上の考慮事項

永続状態情報の処理とは、オブジェクトの活性化中または活性化後のある時点でディスクから永続状態情報を読み取り、必要な場合、非活性化中または非活性化後のある時点でそれを書き込むことです。Basic サンプル・アプリケーションでは、次の2つのオブジェクトが永続状態情報を処理します。

- Registrar オブジェクト

■ CourseSynopsisEnumerator オブジェクト

以降の2つの節では、この2つのオブジェクトが永続状態情報を処理する方法に関する設計上の考慮事項を説明します。

Registrar オブジェクト

Registrar オブジェクトのオペレーションの1つによって、クライアント・アプリケーションにコース情報の詳細が返されます。一般的なシナリオでは、数多くのコース概要を参照した学生は、通常2つか3つのコースに関する詳細情報を一度に参照しようとします。

このシナリオを効率的にインプリメントするために、Registrar オブジェクトには `get_course_details()` オペレーションが定義されています。このオペレーションは、コース番号のリストを指定するパラメータを入力として受け付けます。次にこのオペレーションは、データベースからコースの詳細を検索し、その詳細をクライアント・アプリケーションに返します。このオペレーションがインプリメントされるオブジェクトはプロセス・パウンドなので、このオペレーションはその呼び出しの完了後に状態データをメモリに保持するのを避ける必要があります。

Registrar オブジェクトは、永続状態をメモリに保持しません。クライアント・アプリケーションが `get_course_details()` オペレーションを呼び出すと、このオブジェクトは関連するコース情報を University データベースからフェッチしてクライアントに送信します。このオブジェクトは、コース・データをメモリ内に保持しません。このオブジェクトの `activate_object()` オペレーションと `deactivate_object()` オペレーションでは永続状態は処理されません。

CourseSynopsisEnumerator オブジェクト

CourseSynopsisEnumerator オブジェクトは、自身が University データベースから検索するコース概要を処理します。状態処理に関する設計上の考慮事項は、ディスクから状態を読み取る方法です。このオブジェクトは、状態をディスクに書き込みません。

CourseSynopsisEnumerator オブジェクトの機能には、次の3つの重要な側面があります。これらの側面は、このオブジェクトがその永続状態を読み取る方法に影響を与えます。

- このオブジェクトの OID には、コース概要を取得するための最初のクライアント要求で指定される検索基準が含まれます。この検索基準は、データベースのキーとして機能します。つまり、このオブジェクトは OID に格納されている検索基準に基づいてデータベースから情報を抽出します。
- このオブジェクトのすべてのオペレーションは、そのオブジェクトがメモリに読み込むコース概要を使用します。
- このオブジェクトは、非活性化されるときにコース概要をメモリからフラッシュする必要があります。

この3つの側面を考えたとき、このオブジェクトでは次のことを行うのが合理的です。

- 活性化時に、そのオブジェクトの `activate_object()` オペレーションを介してその永続状態情報を読み取ります。
- 非活性化時に、`deactivate_object()` オペレーションを介してメモリからコース概要をフラッシュします。

したがって、`CourseSynopsisEnumerator` オブジェクトが活性化されるときに、そのオブジェクトの `activate_object()` オペレーションは次のことを行います。

1. その OID から検索基準を抽出します。
2. 検索基準に一致するコース概要をデータベースから検索します。

注記 オブジェクトの `Tobj_ServantBase::activate_object()` オペレーションまたは `Tobj_ServantBase::deactivate_object()` オペレーションをインプリメントする場合は、インプリメンテーション・ヘッダ・ファイル（つまり、`application_i.h` ファイル）を編集し、これらのオペレーションの定義をそのオブジェクトのインターフェイスのクラス定義テンプレートに追加してください。

University データベースの使い方

University サンプル・アプリケーションが University データベースを使用する方法について、次のことに注意してください。

- すべての University サンプル・アプリケーションは、University データベースにアクセスしてコース情報と学生情報を操作します。通常、これはインプリメンテーション・ファイルに記述するコードの大きい部分を占めます。University サンプルのインプリメンテーション・ファイルをシンプルにし、データベース・コードの代わりに CORBA 機能に集中できるようにするために、このサンプルではデータベースに対する読み書きを行うすべてのコードが1つのクラス・セットにラップされています。`utils` ディレクトリ内の `samplesdb.h` ファイルには、これらのクラスの定義が記述されています。これらのクラスは、University データベース内のコース・レコードと学生レコードを読み書きするために必要なすべての SQL 呼び出しを行います。

注記 Wrapper および Production サンプル・アプリケーションの BEA Tuxedo Teller アプリケーションは、University データベース内の口座情報に直接アクセスし、`samplesdb.h` ファイルを使用しません。

Basic サーバ・アプリケーションに組み込むファイルの詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。

- `CourseSynopsisEnumerator` オブジェクトは、データベース・カーソルを使用して University データベースから一致するコース概要を検索します。データベース・カーソルは複数のトランザクションにまたがることのできないので、`CourseSynopsisEnumerator` オブジェクトの `activate_object()` オペレーションは一致するすべてのコース概要をメモリに読み取ります。カーソルはイテレータ・クラスによって管理されているので、`CourseSynopsisEnumerator` オブジェクトからは見えないことに注意してください。University サンプル・アプリケーションがトランザクションを使用する方法については、「第 6 章 トランザクションの CORBA サーバ・アプリケーションへの統合」を参照してください。

Basic サンプル・アプリケーションがデザイン・パターンを適用する方法

Basic サンプル・アプリケーションは、次のデザイン・パターンを使用します。

- Process-Entity
- List-Enumerator

この節では、この2つのデザイン・パターンが Basic サンプル・アプリケーションに適している理由と、Basic サンプル・アプリケーションがこれらのパターンをインプリメントする方法について説明します。

Process-Entity デザイン・パターン

第1章の30ページ「Process-Entity デザイン・パターン」で説明したように、このデザイン・パターンは、クライアント・アプリケーションによって必要とされるデータ・エンティティを処理する1つのプロセス・オブジェクトが存在する場合に適しています。データ・エンティティは、クライアント・アプリケーションではなくこのプロセス・オブジェクトによって操作される CORBA `struct` としてカプセル化されます。

Process-Entity デザイン・パターンを Basic サンプル・アプリケーションに適用すると、Basic サンプル・アプリケーションは細粒度オブジェクトのインプリメントを回避できます。たとえば、`Registrar` オブジェクトは、同じような多数のコース・オブジェクト・セットの有効な代替となります。単一の粗粒度 `Registrar` オブジェクトを管理する負荷は、何百、何千もの細粒度のコース・オブジェクトを管理する負荷に比べれば、それほど大きくはありません。

Process-Entity デザイン・パターンの詳細については、『デザイン・パターンに関する技術情報』を参照してください。

List-Enumerator デザイン・パターン

このデザイン・パターンは、大きすぎて1つの応答でクライアント・アプリケーションに返すことができないデータの内部リストをオブジェクトが生成した場合に適しています。このため、オブジェクトは1回の応答で最初のデータ・リストをクライアント・アプリケーションに返し、後続の応答で残りのデータを返すことができなければなりません。

また、List-Enumerator オブジェクトは、既に返されたデータの量を同時に追跡して、後続のリストを正確に返すことができるようにする必要があります。List-Enumerator オブジェクトは常に状態を持ち（クライアント呼び出しにまたがって活性化され、メモリに常駐する）、サーバ・アプリケーションはそれらが不要になったときにそれらを非活性化できます。

List-Enumerator デザイン・パターンは、CourseSynopsisEnumerator オブジェクトに最適です。このデザイン・パターンをインプリメントすると、次のメリットが得られます。

- University サーバ・アプリケーションは、大きいコース概要リストを、クライアント・アプリケーションが処理できる方法、つまり管理可能な分量で返すための方法を手に入れます。
- 各 CourseSynopsisEnumerator オブジェクトは一意で、その内容はこのオブジェクトが作成される原因となった要求によって決まります。また、各 CourseSynopsisEnumerator のオブジェクト ID も一意です。クライアントが Registrar オブジェクトの `get_courses_synopsis()` オペレーションを呼び出すと、Registrar オブジェクトは次のものを返します。
 - 最初の概要リスト
 - 残りの概要を返すことができる CourseSynopsisEnumerator オブジェクトのオブジェクト・リファレンス

このため、後続のすべての呼び出しは適切な CourseSynopsisEnumerator オブジェクトに対して行われます。これは、サーバ・プロセスに CourseSynopsisEnumerator クラスのアクティブ・インスタンスが複数存在する場合に重要です。

`get_courses_synopsis()` オペレーションは一意の CourseSynopsisEnumerator オブジェクト・リファレンスを返すので、クライアント要求どうしが衝突することはありません。つまり、クライアント要求が間違った CourseSynopsisEnumerator オブジェクトに誤って送られることはありません。

Registrar オブジェクトは `get_courses_synopsis()` オペレーションを備えています。データベース問い合わせと概要リストの知識はすべて CourseSynopsisEnumerator オブジェクトに埋め込まれます。この場合、Registrar オブジェクトは単にクライアントが次のデータを取得するための手段として機能します。

- 最初の概要リスト
- 残りの概要を返すことができる `CourseSynopsisEnumerator` オブジェクトのリファレンス

BEA Tuxedo システムに組み込まれた性能効率化

BEA Tuxedo システムは、同じサーバ・プロセス内の 2 つのオブジェクト間のデータ・マーシャリングを自動的に無効化することによって、性能の効率化を実現します。この効率化は、次の環境が存在するときに得られます。

- オブジェクト・リファレンスが、自身が作成されたサーバ・プロセスを含んでいるグループに送られる場合。
- そのサーバ・プロセス内のあるオブジェクトがそのオブジェクト・リファレンスを使用してオペレーションを呼び出し、それによって同じプロセスでオブジェクトがインスタンス化される場合

この例には、`Registrar` オブジェクトが `CourseSynopsisEnumerator` オブジェクトのオブジェクト・リファレンスを作成し、それによってそのオブジェクトがインスタンス化された場合などがあります。この 2 つのオブジェクト間の要求と応答では、データ・マーシャリングは行われません。

状態を持つオブジェクトの事前活性化

状態を持つオブジェクトの事前活性化機能を使用すると、クライアント・アプリケーションがオブジェクトを呼び出す前にそのオブジェクトを活性化できます。この機能は、University サンプルの `CourseSynopsisEnumerator` オブジェクトなどのイテレータ・オブジェクトを作成するときに特に役立ちます。

状態を持つオブジェクトの事前活性化では、

`TP::create_active_object_reference()` オペレーションを使用します。通常、クライアントがあるオブジェクトに対する呼び出しを発行するまで、CORBA サーバ・アプリケーションでそのオブジェクトは作成されません。

ただし、オブジェクトを事前活性化し、

`TP::create_active_object_reference()` オペレーションを使用してそのオ

プロジェクトのリファレンスをクライアントに受け渡すことによって、クライアント・アプリケーションは既に活性化され、状態を持っているオブジェクトを呼び出すことができます。

注記 状態を持つオブジェクトの事前活性化の機能は、WebLogic Enterprise バージョン 4.2 で最初に導入されました。

状態を持つオブジェクトを事前活性化する方法

事前活性化の機能を使用するためのプロセスは、サーバ・アプリケーションに次のコードを記述することです。

1. オブジェクトを作成するための C++ `new` 文の呼び出しを含めます。
2. オブジェクトの状態を設定します。
3. `TP::create_active_object_reference()` オペレーションを呼び出して、新しく作成されたオブジェクトのリファレンスを取得します。このオブジェクト・リファレンスは、クライアント・アプリケーションに返すことができます。

このように、事前活性化オブジェクトは、TP フレームワークがそのオブジェクトの `Server::create_servant()` オペレーションと `Tobj_ServantBase::activate_object()` オペレーションを呼び出さずに作成されます。

事前活性化オブジェクトの使い方に関する注意

事前活性化機能を使用するときには、以下のことに注意してください。

- 事前活性化オブジェクトには、`process` 活性化方針が割り当てられている必要があります。このため、これらのオブジェクトは、プロセスの終了時か、それらのオブジェクトの `TP::deactivateEnable()` オペレーションの呼び出しによってのみ非活性化できます。
- `TP::create_active_object_reference()` オペレーションによって作成されるオブジェクト・リファレンスは一時的なものです。これは、事前活性化オブジェクトは自身が作成されたプロセスの存続期間にわたって存在し、別のサーバ・プロセスで再び活性化してはならないからです。

3 Basic CORBA サーバ・アプリケーションの設計とインプリメント

ある一時オブジェクト・リファレンスが作成されたプロセスのシャットダウン後にクライアント・アプリケーションがそのオブジェクト・リファレンスを呼び出した場合、TP フレームワークは次の例外を返します。

```
CORBA::OBJECT_NOT_EXIST
```

- 事前活性化されるオブジェクトの場合、通常その状態はクラッシュが発生したときに回復できません。ただし、これは大きな問題ではありません。このようなオブジェクトは、一般に特定の一連のオペレーションのコンテキスト内で使用され、その後削除されるからです。オブジェクトの状態は、それらのオペレーションの外部では意味を持ちません。

サーバがクラッシュし、そのときに削除されたオブジェクトをクライアント・アプリケーションが呼び出そうとする状況を回避するには、事前活性化されるオブジェクトに対する

```
Tobj_ServantBase::activate_object()
```

 オペレーションのインプリメンテーションに `TobjS::ActivateObjectFailed` 例外を追加します。クライアントがサーバ・クラッシュ後にこのようなオブジェクトを呼び出そうとし、TP フレームワークがそのオブジェクトの

```
Tobj_ServantBase::activate_object()
```

 オペレーションを呼び出した場合、TP フレームワークはクライアント・アプリケーションに次の例外を返します。

```
CORBA::OBJECT_NOT_EXIST
```

- 事前活性化はあまり使用しないでください。すべてのプロセス・バウンド・オブジェクトと同じように、事前活性化によって少ないリソースが事前に割り当てられるからです。

4 マルチスレッド CORBA サーバ・ア プリケーションの作 成

ここでは、以下の内容について説明します。

- 概要
- マルチスレッド CORBA サーバ・アプリケーションの開発とビルド
- マルチスレッド simpapp サンプル・アプリケーションのビルドと実行
- マルチスレッド CORBA サーバ・アプリケーションの管理

概要

ここでは、以下の内容について説明します。

- はじめに
- マルチスレッド CORBA サーバをサポートするためのメカニズム
- マルチスレッド・システムでのシングル・スレッド・サーバ・アプリケーションの実行

はじめに

複数の独立したスレッドを使用するようにアプリケーションを設計すると、アプリケーション内に並行性がもたらされ、総合的なスループットを改善できます。複数のスレッドを使用すると、各スレッドが複数の独立したタスクを並列に処理する効率的なアプリケーションを構築できます。マルチスレッドは、以下の場合に特に役立ちます。

- ほかの処理に必ずしも依存しない一連の長いオペレーションが存在する。
- 共有されるデータの量が少なく、識別可能である。
- 並列に実行できる複数のアクティビティにタスクを分割できる。
- オブジェクトがリエントラントでなければならないときが存在する。

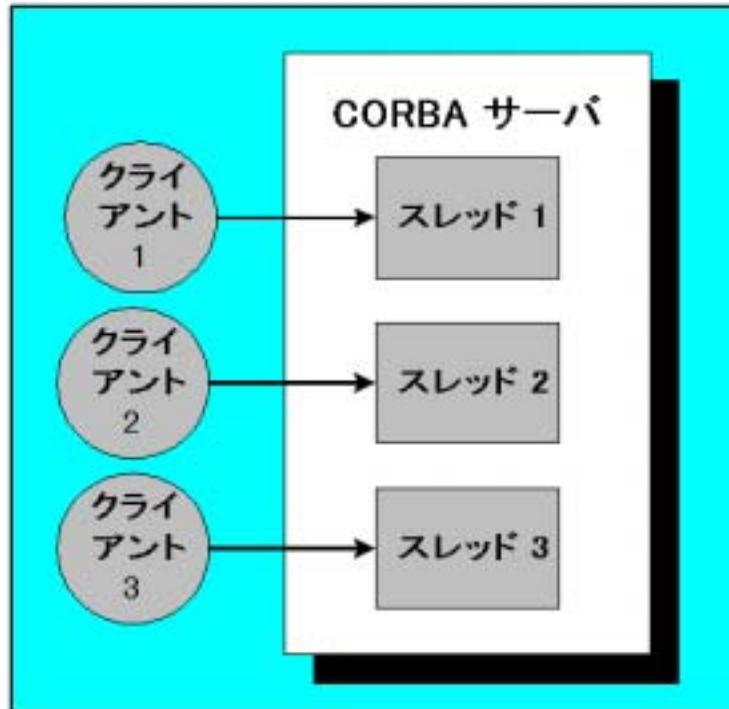
歴史的に見て、業界レベルのマルチスレッド・アプリケーションの設計とインプリメントは複雑なものでした。BEA Tuxedo が提供するサポートは、スレッドを CORBA サーバ環境の内部で管理することによって、こうした複雑さを単純化します。

BEA Tuxedo ソフトウェアは、次のマルチスレッド特性を備えたサーバ・アプリケーションをサポートします (図 4-1 を参照)。

- サーバ・オブジェクトのインスタンスが複数のクライアント要求を同時に処理できる。

- サーバ・オブジェクトが自身に対する再帰的呼び出しを行うことができる。
- サーバ・オブジェクトが独自のスレッドを作成および監視して、サーバント・メソッドの中で並列処理をインプリメントできる。

図 4-1 マルチスレッド CORBA サーバ・アプリケーション



通常、BEA Tuxedo ソフトウェアは、サーバ・アプリケーションの代わりにスレッドを作成および管理します。マルチスレッド・サーバ・アプリケーションをビルドする場合、TP フレームワークの使い方、サーバントのインプリメント方法、および独自のスレッドを作成するオブジェクトの設計方法が変わります。

4 マルチスレッド CORBA サーバ・アプリケーションの作成

BEA Tuxedo ソフトウェアを使用すると、要求ごとのスレッド・モデルか、オブジェクトごとのスレッド・モデルのいずれかをインプリメントできます。各モデルについては、「第 4 章の 28 ページ「スレッド・モデル」」で説明します。

要件、目標、概念

コンピュータ・オペレーションの中には、完了するのに長い時間を要するものがあります。マルチスレッド設計では、オペレーションの要求と完了の間の待機時間を飛躍的に短縮できます。これは、オペレーションが数多くの I/O オペレーションを実行する環境に当てはまります。たとえば、データベースにアクセスするとき、リモート・オブジェクトのオペレーションを呼び出すとき、マルチ・プロセッサ・マシン上の CPU バウンドなどです。サーバ・プロセスでマルチスレッドをインプリメントすると、サーバが一定時間に処理できる要求の数が増加します。

マルチスレッド・サーバ・アプリケーションの主要な要件は、複数のクライアント要求を同時に処理することです。この種のサーバを開発する目的は次のとおりです。

- プログラム設計の単純化

これは、既存のプログラミング抽象化を使用して複数のサーバ・タスクを独立して実行することにより達成されます。

- スループットの向上

これは、マルチプロセッサ・ハードウェア・プラットフォームの並列処理のメリットを活用し、計算処理と通信をオーバーラップさせることによって達成されます。

- 知覚応答時間の向上

独立したスレッドを異なるサーバ・タスクに関連付けることによって、クライアントは長時間にわたって互いをブロックすることがなくなります。

- リモート・プロシージャ・コールと会話のコーディングの単純化

独立したスレッドを使用して異なるリモート・プロシージャ・コール (RPC) と会話とのやり取りを実現すると、一部のアプリケーションのコーディングが簡単になります。

- 複数アプリケーションへの同時アクセス

レガシー・アプリケーションまたはレガシー・データベースを CORBA サーバ内でラップすると、インプリメンテーションは一度に複数のレガシー・アプリケーションとやり取りできます。

- 必要なサーバ数の削減

1つのサーバで複数のサービス・スレッドをディスパッチできるので、アプリケーションで必要なサーバの数を減らすことができます。

ただし、マルチスレッド設計にはコストがかかります。通常、マルチスレッド・サーバ・アプリケーションでは、シングル・スレッド・サーバより複雑な同期手法が必要になります。アプリケーション開発者は、スレッド・セーフなコードを記述する必要があります。また、スレッドを作成して要求を処理するオーバーヘッドが並列処理のメリットより大きくなる場合もあります。特定の同時実行モデルの実際の性能は、次の要因によって決まります。

- クライアントからの要求の特性

要求の期間は長いですか、それとも短いですか。

- スレッドのインプリメント方法

スレッドはオペレーティング・システム・カーネル、ユーザ領域のライブラリ、両方の組み合わせのいずれで管理されますか。

- オペレーティング・システムとネットワークのオーバーヘッド

接続を繰り返しセットアップおよび切断することによって発生するオーバーヘッドはどのくらいですか。

- 高度なシステム・コンフィギュレーション要因

複製、動的ロード・バランシング、またはほかの要因が性能に影響を与えますか。

スレッド・ライブラリは同時実行モデルを作成するためのメカニズムを提供しますが、そのメカニズムを適切に使用方法を最終的に理解するのは開発者です。デザイン・パターンを学習することによって、アプリケーション開発者は微妙な差異を理解し、さまざまな状態に応じた設計を選択できるようになります。

スレッド・モデル

サーバ内の並行性を設計するために使用できるモデルはいくつか存在します。以降の節では、要求ごとのスレッド・モデル、オブジェクトごとのスレッド・モデル、スレッド・プール、および BEA Tuxedo ソフトウェアが各モデルをインプリメントする方法について説明します。特定のサーバは、要求ごとのスレッド・モデルかオブジェクトごとのスレッド・モデルのいずれかに対応するよう設計されます。

要求ごとのスレッド・モデル

このモデルでは、クライアントからの各要求は別々の制御スレッドで処理されます。このモデルは、サーバが一般に複数のクライアントから長期の要求を受信するときに役立ちます。短期の要求の場合、要求ごとに新しいスレッドを作成するオーバーヘッドが大きくなるので、あまり役立ちません。新しい要求が到着するたびに、BEA Tuxedo はその要求をスレッドに関連付けて、その要求を実行します。マルチスレッド・アプリケーション・サーバ・プロセスは一度に複数のスレッドをホストできるので、一度に複数のクライアント要求を同時に実行できます。BEA Tuxedo は、要求とスレッドの関連付けを制御します。このため、アプリケーションは BEA Tuxedo より強力な制御を必要としない限り、スレッドを明示的に作成する必要がありません。

要求ごとのスレッド・モデルでは、アプリケーション・サーバをスレッド・セーフにする必要があります。つまり、複数のサーバ・オブジェクト間で共有されるデータへのアクセスを制御するための同時実行メカニズムをインプリメントする必要があります。同時実行制御メカニズムを使用しなければならぬ場合、アプリケーション開発プロセスが複雑化します。さらに、多数のクライアントが同時に要求を行った場合、この設計ではオペレーティング・システム・リソースが大量に消費される可能性があります。

オブジェクトごとのスレッド・モデル

オブジェクトごとのスレッド・モデルでは、サーバ・プロセス内の活性化された各オブジェクトが常に 1 つのスレッドに関連付けられます。オブジェクトに対する要求ごとに、ディスパッチ・スレッドとオブジェクトとの関連付けが確立されます。同じオブジェクトに対する連続する要求は、別個のスレッドによって処理できます。特定のスレッドを複数のオブジェクトで共有できます。

スレッド・プール

スレッド・プールは、スレッドを管理するコストを削減するための手段です。起動時に必要に応じてスレッドの作成、割り当て、および解放を行うプール。スレッドは、次の要求の処理に必要となるまでプールで待機します。スレッド・プールを使用すると、前述のスレッド・モデルをサポートできます。たとえば、要求ごとのスレッド・モデルで要求に対してスレッドを割り当て、メソッドの実行に使用して、プールに戻すことができます。

スレッドの割り当てと割り当て解除には、特に要求とオブジェクトの存続期間が短い場合、時間とコストがかかるおそれがあります。スレッド・プールは、スレッドを管理するコストを削減するための手段です。起動時に、または必要に応じて、BEA Tuxedo ソフトウェアは利用可能なスレッドのプールにスレッドを作成し、そのプールからスレッドを割り当てて、そのプールにスレッドを戻します。スレッドはプールに存在し、以後の要求の処理に必要となるまで待機します。

アプリケーション・サーバ・プロセス用の BEA Tuxedo スレッド・プールの初期サイズと最大サイズは、サーバ・コンフィギュレーション・ファイルの設定値によって制御されます。起動時に、最小プール・サイズが事前に割り当てられます。処理する要求が到着すると、BEA Tuxedo ソフトウェアはプールからスレッドを割り当ててその要求を処理します。要求の処理に利用可能なスレッドがプールに存在せず、プールに空きがある場合、BEA Tuxedo ソフトウェアは新しいスレッドを作成してその要求を処理します。要求が到着したときに利用可能なスレッドがプールに存在せず、新しいスレッドを作成できない場合、その要求はスレッドが利用可能になるまでキューに格納されます。

スレッド・プールは、サーバ・スレッドのために消費されるシステム・リソースの量を制限したい場合に適しています。スレッド・プールを使用すると、同時要求の数がプール内のスレッドの数を上回るまでクライアント要求が同時に実行されます。

BEA Tuxedo スレッド・プールには、次の特性と振る舞いがあります。

- プールの最大サイズを BEA Tuxedo 管理機能として設定できます。アプリケーション自体に変更を加えずにプールのサイズを調節できます。
- BEA Tuxedo ソフトウェアは、必要に応じてプールからスレッドを割り当てます。スレッドは要求が処理される間使用され、その後プールに戻されます。

- スレッドは、複数の要求とオブジェクトを処理するために連続的に再利用できます。

リエントラント・サーバント

BEA Tuxedo ソフトウェアは、オブジェクトが自身のオペレーションを再帰的に呼び出すための機能を提供します。この機能を使用するときには、オブジェクトをインプリメントする方法に十分な注意を払う必要があります。アプリケーション・コードは、共有される状態データへのアクセスの制御に必要なオペレーティング・システム同時実行メカニズムを採用する必要があるからです。Process または Distribution Adapter デザイン・パターンをインプリメントするオブジェクトを使用するケースなど、オブジェクトの共有状態がほとんどまたはまったく存在せず、リエントラントをサポートするのが比較的簡単な場合があります。

また、BEA Tuxedo ソフトウェアを使用すると、活性化されたオブジェクトのメソッド呼び出しのリエントラントを有効化または禁止することができます。リエントラントはデフォルトによって無効化されています。活性化されたオブジェクトに対する要求が到着したときに、そのオブジェクトが異なるスレッドで別の要求を実行している場合、次の規則が適用されます。

- `_is_reentrant` メソッドが `TRUE` を返した場合、新しいスレッドがプールから割り当てられ、同じサーバント・インスタンスを使用して適切なメソッドに要求がディスパッチされます。複数のスレッドがオブジェクトと対話する場合にオブジェクトの状態の整合性を保証するのは、サーバント・インプリメンテーション・コードの責任です。
- `_is_reentrant` メソッドが `FALSE` を返した場合、サーバントの新しいインスタンスが作成され、メソッドが新しいインスタンスにディスパッチされます。このインスタンスは自動的に削除されません。以後のリエントラント要求はいずれかのインスタンスにディスパッチされます。

注記 リエントラント・サーバント・メカニズムは、`PER_REQUEST` 同時実行手法が指定された状態でサーバが起動した場合にのみ利用可能です。

このメソッドの使い方については、『BEA Tuxedo CORBA プログラミング・リファレンス』を参照してください。

Current オブジェクト

マルチスレッド CORBA サーバ・アプリケーション環境の最も重要な属性の 1 つは、Current オブジェクトを適切に使用および管理できることです。このため、次のような振る舞いが保証されます。

- 個々のスレッドは、適切なトランザクション・コンテキストとセキュリティ・コンテキストの中で機能します。
- Current オブジェクトは、異なるスレッドからアクセスされたときに適切に振る舞います。

BEA Tuxedo 製品は、OMG 発行の ORB Portability 仕様によって定義されているマルチスレッド・モデルに準拠しています。この仕様は、OMG CORBA 仕様に組み込まれています。BEA Tuxedo 製品では、CORBA::Current から継承されたインターフェイスのオペレーションは、Current オブジェクトが取得されたスレッドに関連付けられている状態ではなく、オペレーションが呼び出されるスレッドに関連付けられている状態にアクセスできます。この振る舞いの理由は次の 2 つです。

- あるスレッドが別のスレッドの状態を操作するのを防ぐため
- メソッドごとにスレッド・コンテキスト内で新しい Current オブジェクトを取得してナロー変換する必要性をなくするため

マルチスレッド環境で使用する場合、次のオブジェクトの振る舞いは ORB Portability 仕様に合致します。

- CosTransactions::Current
- SecurityLevel1::Current
- SecurityLevel2::Current
- PortableServer::Current

たとえば、アプリケーションがあるスレッドから別のスレッドにトランザクションを受け渡す場合、そのアプリケーションは CosTransactions::Current オブジェクトを使用してはなりません。代わりに、アプリケーションはほかのスレッドに CosTransactions::Control オブジェクトを受け渡します。CosTransactions::Current オブジェクトを受け渡すと、受信側のスレッドはそのスレッドに関連付けられているトランザクション状態だけにアクセスできます。

マルチスレッド CORBA サーバをサポートするためのメカニズム

この節では、BEA Tuxedo CORBA に用意されているマルチスレッド・サーバ・アプリケーションをサポートする以下のツール、API、および管理機能の概要について説明します。

- コンテキスト・サービス
- TP フレームワークのクラスとメソッド
- build コマンドの機能
- 管理用ツール

コンテキスト・サービス

オブジェクト・インプリメンテーションに独自のスレッドを作成して管理することを選択できます。ほかのスレッドは、BEA Tuxedo CORBA ソフトウェアによって自動的に管理されます。The BEA Tuxedo CORBA ソフトウェアは、作成および管理する各スレッドのコンテキスト情報を内部に保持します。この必須コンテキスト情報は、CORBA 要求の処理中に使用されます。BEA Tuxedo CORBA は、アプリケーションがいつ独自のスレッドを作成して削除するかに関する知識を持ちません。このため、プログラマはコンテキスト・サービス・メカニズムを使用して、BEA Tuxedo サービスの呼び出し前に独自のスレッドを適切に初期化し、スレッドの削除時に不要になったコンテキスト・リソースを解放できます。

次の ORB メソッド・セットは、スレッド管理の要件を満たします。これらのメソッドを、まとめてコンテキスト・サービスと呼びます。

- `ORB::get_ctx()`

オブジェクトがスレッドを作成するときに、オブジェクトは ORB のこのオペレーションを呼び出して、オブジェクトがスレッドに受け渡すことができるシステム・コンテキスト情報を取得します。このオペレーションは、既にコンテキストを持っているスレッドから呼び出される必要があります。たとえば、メソッドがディスパッチされたスレッドはコンテキストを持っています。このオペレーションの使い方については、『BEA

Tuxedo CORBA プログラミング・リファレンス』の「ORB::get_ctx()」を参照してください。

■ ORB::set_ctx()

オブジェクトがスレッドを作成すると、そのスレッドは通常 get_ctx メソッドを呼び出したスレッドからコンテキスト情報を取得します。作成されたスレッドは、ORB::set_ctx を呼び出してそのスレッドが動作する必要があるシステム・コンテキストを設定するときに、そのコンテキストを使用します。このオペレーションの使い方については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「ORB::set_ctx()」を参照してください。

■ ORB::clear_ctx()

作成されたスレッドは、自己の仕事を完了するとこのメソッドを呼び出して自身とシステム・コンテキストとの関連付けを解除します。このオペレーションの使い方については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「ORB::clear_ctx()」を参照してください。

■ ORB::inform_thread_exit()

スレッドは、自己の仕事が完了するとこのメソッドを呼び出して、BEA Tuxedo システムにアプリケーション管理スレッドに関連付けられているリソースを解放できることを通知します。このオペレーションの使い方については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「ORB::inform_thread_exit()」を参照してください。

TP フレームワークのクラスとメソッド

BEA Tuxedo TP フレームワークの次のクラスとメソッドは、マルチスレッド・サーバ・アプリケーションをサポートします。

■ ServerBase クラス

ServerBase クラスのデフォルト・インプリメンテーションを上書きするために、アプリケーション開発者は ServerBase から継承されたクラスを作成できます。既にサポートされている ServerBase メソッドのほかに、次のメソッドがマルチスレッド・サーバ・アプリケーションをサポートします。

- create_servant_with_id()
- thread_initialize()

4 マルチスレッド CORBA サーバ・アプリケーションの作成

- `thread_release()`

これらのメソッドを使用すると、アプリケーションのマルチスレッド特性を細かく制御できます。これらのメソッドの使い方については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「ServerBase インターフェイス」を参照してください。

■ Tobj_ServantBase クラス

このクラスは、マルチスレッド・サーバ・アプリケーションをサポートする次のメソッドを提供します。

- `Tobj_ServantBase::_is_reentrant()`
- `Tobj_ServantBase::_add_ref()`
- `Tobj_ServantBase::_remove_ref()`

これらのメソッドの詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「Tobj_ServantBase インターフェイス」を参照してください。

build コマンドの機能

`buildobjserver` コマンドと `buildobjclient` コマンドには、次のスレッド管理機能が組み込まれています。

- `buildobjserver` コマンドには、プラットフォーム固有のスレッド・ライブラリ・サポートが組み込まれています。このため、サーバ・アプリケーションは BEA Tuxedo ソフトウェアのマルチスレッド・サポートとの互換性を保持します。

`buildobjserver` コマンドには、マルチスレッド・サーバ・アプリケーションまたはシングル・スレッド・サーバ・アプリケーションをビルドするためのコマンド行オプションが組み込まれています。

- `buildobjclient` コマンドには、プラットフォーム固有のスレッド・ライブラリ・サポートが組み込まれています。このため、クライアント・アプリケーションは BEA Tuxedo ソフトウェアのマルチスレッド・サポートとの互換性を保持します。

管理用ツール

BEA Tuxedo システムは、アプリケーションを開発および実行するためのコンフィギュレーション・ファイルを使用します。通常、アプリケーション開発者がこれらのファイルを作成し、BEA Tuxedo システム管理者が必要に応じてその内容を修正し、アプリケーションとシステムの要件を満たします。

スレッドのサポートに関連する制御パラメータでは、次のことを指定します。

- サーバをシングル・スレッドにするか、マルチスレッドにするか
- オブジェクトのメソッドをディスパッチするためのスレッド・プールのサイズ

UBBCONFIG ファイルのスレッド・パラメータの詳細については、第 4 章の 67 ページ「UBBCONFIG ファイルの例」を参照してください。

マルチスレッド・システムでのシングル・スレッド・サーバ・アプリケーションの実行

BEA Tuxedo CORBA のスレッド・サポートのデフォルトの振る舞いは、シングル・スレッド・サーバ・サポート環境をエミュレートすることです。マルチスレッド環境でシングル・スレッド CORBA アプリケーションを実行する場合は、サーバ・アプリケーション・コードとコンフィギュレーション・ファイルを変更する必要がありません。ただし、既存のシングル・スレッド・アプリケーションを実行する前に、`buildobjserver` コマンドと `buildobjclient` コマンドを使用してそのアプリケーションを再ビルドする必要があります。サーバ・アプリケーションのマルチスレッドを有効化しなかった場合、そのアプリケーションはシングル・スレッド・サーバとして動作します。

マルチスレッド CORBA サーバ・アプリケーションの開発とビルド

ここでは、以下の内容について説明します。

- `buildobjserver` コマンドの使い方
- `buildobjclient` コマンドの使い方
- 非リエントラント・サーバントの作成
- リエントラント・サーバントの作成
- マルチスレッド `simpapp` サンプル・アプリケーションのビルドと実行

`buildobjserver` コマンドの使い方

`buildobjserver` コマンドは、次の機能を通じて CORBA サーバ・アプリケーションをサポートします。

- プラットフォーム固有のスレッド・ライブラリ
- マルチスレッド・サポートの指定
- 代替サーバ・クラスの指定

プラットフォーム固有のスレッド・ライブラリ

`buildobjserver` コマンドによって生成されるサーバ・アプリケーションは、適切なプラットフォーム固有のコンパイラ設定を使用してコンパイルされ、適切なプラットフォーム固有のスレッド・サポート・ライブラリを使用してリンクされます。これにより、BEA Tuxedo ソフトウェアが提供する共有ライブラリとの互換性が保証されます。

マルチスレッド・サポートの指定

マルチスレッドをサポートする CORBA サーバ・アプリケーションを作成する場合、アプリケーションをビルドするときに `buildobjserver` コマンドで `-t` オプションを指定する必要があります。実行時に、BEA Tuxedo システムは、実行可能プログラムと、CORBA サーバ・アプリケーションのコンフィギュレーション・ファイル `UBBCONFIG` で選択されたスレッド・モデルとの互換性を検証します。`UBBCONFIG` ファイルでスレッド・モデルを設定する方法については、第 4 章の 67 ページ「`UBBCONFIG` ファイルの例」を参照してください。

注記 CORBA サーバ・アプリケーションのビルド時に `-t` を指定する場合、`UBBCONFIG` ファイルの `MAXDISPATCHTHREADS` パラメータを 1 より大きい値に設定する必要があります。それ以外の値を設定すると、CORBA サーバ・アプリケーションはシングル・スレッド・サーバとして動作することになります。

注記 マルチスレッド共同クライアント/サーバ・インプリメンテーションはサポートされていません。

コンフィギュレーション・ファイルに互換性のないスレッド・モデルを指定してシングル・スレッドの実行可能プログラムを起動しようとすると、次のイベントが発生します。

- BEA Tuxedo ソフトウェアは、ログ・ファイルに警告を記録します。
- サーバの実行可能プログラムがシングル・スレッド・サーバとして起動します。

代替サーバ・クラスの指定

`ServerBase` クラスから継承された独自の `Server` クラスを実装する場合、`buildobjserver` コマンドで `-b` オプションを使用してその代替 `Server` クラスを指定する必要があります。`buildobjserver` コマンドは、次の構文で `-b` オプションをサポートしています。

```
buildobjserver [-v] [-o outfile] [-f {firstfiles|@def-file}]  
[-l {lastfiles|@def-file}] [-r rmname] [-b bootserverclass] [-t]
```

4 マルチスレッド CORBA サーバ・アプリケーションの作成

上の構文中、*bootserverclass* の値は、CORBA サーバ・アプリケーションの起動時に使用される C++ クラスを指定します。-b オプションを指定しない場合、BEA Tuxedo システムは *Server*. という名前でクラスのインスタンスを作成します。

-b オプションを指定する場合、Tuxedo システムは代替サーバ・クラスの実行関数を作成し、プロジェクトは -b オプションで指定した *bootserverclass* の名前を持つヘッダ・ファイルを提供する必要があります。ヘッダ・ファイルには、代替 C++ クラスの定義が含まれます。代替 *Server* クラスは、*ServerBase* クラスから継承されなければなりません。

たとえば、コマンド行で -b *AslanServer* と指定した場合、アプリケーション・プロジェクトは *AslanServer.h* ファイルを提供する必要があります。*AslanServer.h* ファイルは、*bootserverclass.h* ファイルの例です。*bootserverclass* ファイルは、このコード例と同じようなロジックを提供します。

コードリスト 4-1 *bootserverclass.h* ファイルの例

```
// ファイル名 : AslanServer.h
#include <Server.h>
class AslanServer : public ServerBase {
public:
    CORBA::Boolean initialize(int argc, char** argv);
    void release();
    Tobj_Servant create_servant(const char* interfaceName);
    Tobj_Servant create_servant_with_id(const char* interfaceName,
                                       const char* stroid);
    CORBA::Boolean thread_initialize(int argc, char** argv);
    void thread_release();
};
```

buildobjclient コマンドの使い方

buildobjclient コマンドを使用してクライアント・アプリケーションの実行可能プログラムを作成する場合、アプリケーションは適切なプラットフォーム固有のコンパイラ設定を使用してコンパイルされ、使用するオペ

レーティング・システム用の適切なスレッド・サポート・ライブラリを使用してリンクされます。これにより、BEA Tuxedo ソフトウェアが提供する共有ライブラリとクライアントの互換性が保証されます。

非リエントラント・サーバントの作成

BEA Tuxedo CORBA 環境で CORBA サーバ・アプリケーションを実行するには、`buildobjserver` コマンドでそのアプリケーションをビルドしておく必要があります。

`buildobjserver -t` オプションを使用すると、BEA Tuxedo システムに CORBA サーバ・アプリケーションがスレッド・セーフであることを通知できます。`-t` オプションは、アプリケーションが共有コンテキスト・データ、およびスレッド・セーフではないほかのプログラミング構造を使用しないことを示します。スレッド・セーフではないシングル・スレッド・アプリケーションをマルチスレッド環境で実行する場合、データが破損するおそれがあります。

アプリケーションのコンフィギュレーション・ファイルを更新してマルチスレッド・サポートを有効化したにもかかわらず、サーバ・インプリメンテーションがリエントラントをサポートできることをアプリケーション・コードに指定していない場合、次のことに注意してください。

- メソッドは BEA Tuxedo システムによって割り当てられた任意のスレッドで実行されます。
- サーバント・インプリメンテーション・コードは、オブジェクトがその状態に同時アクセスすることを必ずしも防ぎません。ただし、活性化されたサーバントは、一度に 1 つの実行スレッドに限定されます。
- メソッドが特定のスレッドで実行されることを保証できません。特定のスレッドに依存するか関連付けられるストレージは使用しないでください。
- サーバントの `activate_object` メソッドまたは `deactivate_object` メソッドが当初呼び出された要求と同じスレッドで実行されるとは仮定しないでください。

4 マルチスレッド CORBA サーバ・アプリケーションの作成

- 追加のアプリケーション管理スレッドをサーバント・メソッドの中に作成できます。オブジェクト・インプリメンテーションは、スレッドが適切に作成、処理、および破棄されることを保証する必要があります。
- アプリケーション管理スレッドには、ほかのオブジェクトの呼び出しを組み込むことができます。
- 同期化のためにシグナルを使用しないでください。シグナルとスレッドの混在はサポートされていません。

注記 プロセスを終了するための `SIGKILL` シグナルがサポートされていません。シングル・スレッドまたはマルチスレッド・アプリケーションに対する `SIGIO` の使用は、BEA Tuxedo CORBA ではサポートされていません。

- 要求レベルのインターセプタは、そのメソッドによって使用される同じスレッドを通じて BEA Tuxedo CORBA によって呼び出されます。

リエントラント・サーバントの作成

マルチスレッド・リエントラント・サーバントは、次の手順で作成します。

- `buildobjserver` コマンドと `-t` オプションを使用して CORBA サーバ・アプリケーションをビルドし、そのアプリケーションの `UBBCONFIG` サーバ・コンフィギュレーション・ファイルを修正します。
- `TobjServantBase::_is_reentrant` メソッドを使用してリエントラントを有効化することによって、CORBA サーバ・アプリケーション・コードを更新します。
- `UBBCONFIG` ファイルに `CONCURR_STRATEGY = PER_REQUEST` 指定することによって、要求ごとのスレッド・モデルを使用してサーバを起動します。

マルチスレッド・リエントラント・サーバントを作成する場合、そのオブジェクトのインプリメンテーション・コードはそのオブジェクトの状態を保護して、複数のスレッドがそのオブジェクトと対話している間その整合性を保証する必要があります。

クライアント・アプリケーションに関する考慮事項

次に、BEA Tuxedo 環境で実行される CORBA クライアント・アプリケーションに関する考慮事項を挙げます。

- IIOP を使用したマルチスレッド CORBA クライアントがサポートされています。
- マルチスレッド・ネイティブ CORBA クライアントはサポートされていません。
- マルチ・スレッド CORBA クライアントは、単一 Bootstrap オブジェクトに制限されています。
- マルチスレッド CORBA クライアントは、ドメインへの単一ログオンに制限されています。
- スタブ・ベースの呼び出しを使用する CORBA クライアントがサポートされています。

4 マルチスレッド CORBA サーバ・アプリケーションの作成

- 動的起動インターフェイス (DII) を使用する CORBA クライアントはサポートされていません。

マルチスレッド simpapp サンプル・アプリケーションのビルドと実行

ここでは、以下の内容について説明します。

- simpapp マルチスレッド・サンプルについて
- サンプル・アプリケーションの機能
- サンプル・アプリケーションのビルドと実行の方法
- サンプル・アプリケーションのシャットダウン

simpapp マルチスレッド・サンプルについて

BEA Tuxedo ソフトウェアには、クライアント・プログラムと CORBA サーバ・プログラムで構成されるマルチスレッド CORBA サンプル・アプリケーションが用意されています。サーバは、クライアントからアルファベットの文字列を受信し、大文字と小文字の文字列を返します。simpapp_mt のマルチスレッド機能は、並列処理を提供します。この並列処理により、単一のサーバ・プロセスは、複数のオブジェクトまたは単一のオブジェクトに対する複数のクライアントの同時要求を処理できます。

注記 simpapp_mt サンプルのクライアント・アプリケーションは、マルチスレッド・クライアント・アプリケーションではありません。

サンプル・アプリケーションの機能

マルチスレッド・サーバの目的は、1 つまたは複数のクライアントからの要求を並列に処理することです。simpapp_mt サンプル・アプリケーションは、`buildobjserver -t` コマンド行オプションを使用し、`UBBCONFIG` ファイルを使用して同時実行手法を指定することによってマルチスレッド機能を例示する CORBA アプリケーションです。

4 マルチスレッド CORBA サーバ・アプリケーションの作成

simpapp_mt サンプルは、最初に SimplePerObject というサーバ・プロセスを作成し、次に SimplePerRequest というサーバ・プロセスを作成します。クライアントは、最初に SimplePerRequest サーバと通信し、次に SimplePerObject サーバと通信します。

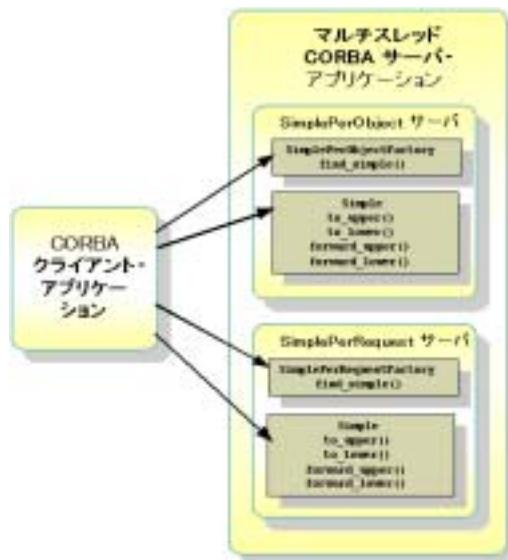
SimplePerRequest の要求ごとのスレッド・サーバ・インプリメンテーションは、スレッド初期化メソッドをインプリメントするユーザ定義のサーバ・クラスの使い方を例示します。SimplePerRequest サーバ・プロセスは、クライアントからの各要求を独立した制御スレッドで処理します。新しい要求が到着するたびに、その要求を処理するためにスレッド・プールからスレッドが割り当てられます。要求が処理され、応答が送信されると、そのスレッドはプールに戻されます。このモデルは、複数のクライアントからの長期の要求を処理するサーバに適しています。

simpapp_mt サンプル・アプリケーションは、次のメソッドを持つ CORBA オブジェクトのインプリメンテーションを提供します。

- to_upper メソッドは、クライアント・アプリケーションからの文字列を受け付けて大文字に変換します。
- to_lower メソッドは、クライアント・アプリケーションからの文字列を受け付けて小文字に変換します。
- forward_upper メソッドは、サーバの別のインスタンスへのアプリケーション管理スレッドを作成し、クライアントから受信した要求をその新しいサーバ・インスタンスに転送して文字列を大文字に変換します。
- forward_lower メソッドは、Simple オブジェクトの別のインスタンスを作成し、クライアントから受信した要求をその新しいインスタンスに転送して文字列を小文字に変換します。

図 4-2 に、オブジェクトごとのスレッド・モデルと要求ごとのスレッド・モデルの両方を使用する simpapp_mt サンプル・アプリケーションのオペレーションを示します。

図 4-2simpapp_mt サンプル・アプリケーション



simpapp マルチスレッド・サンプル・アプリケーションの OMG IDL コード

この章で説明する simpapp マルチスレッド・サンプル・アプリケーションは、次の表に示す CORBA インターフェイスをインプリメントします。

インターフェイス	説明	アクション
SimplePerRequestFactory	Simple オブジェクトのオブジェクト・リファレンスを作成します。	find_simple()
SimplePerObjectFactory	Simple オブジェクトのオブジェクト・リファレンスを作成します。	find_simple()
Simple	文字列の大文字と小文字を変換します。	to_upper() to_lower() forward_upper() forward_lower()

4 マルチスレッド CORBA サーバ・アプリケーションの作成

リスト 4-2 に、simpapp_mt サンプル・アプリケーションの CORBA インターフェイスを定義した simple.idl ファイルの内容を示します。

コード リスト 4-2 simpapp_mt サンプル・アプリケーション用の OMG IDL コード

```
#pragma prefix "beasys.com"

interface Simple
{
    // 文字列を小文字に変換 (新しい文字列を返す)
    string to_lower(in string val);

    // 文字列を大文字に変換 (置換)
    string to_upper(in string val);

    // ほかのサーバを使用して文字列を小文字に変換
    string forward_lower(in string val);

    // ほかのサーバを使用して文字列を大文字に変換
    string forward_upper(in string val);
};

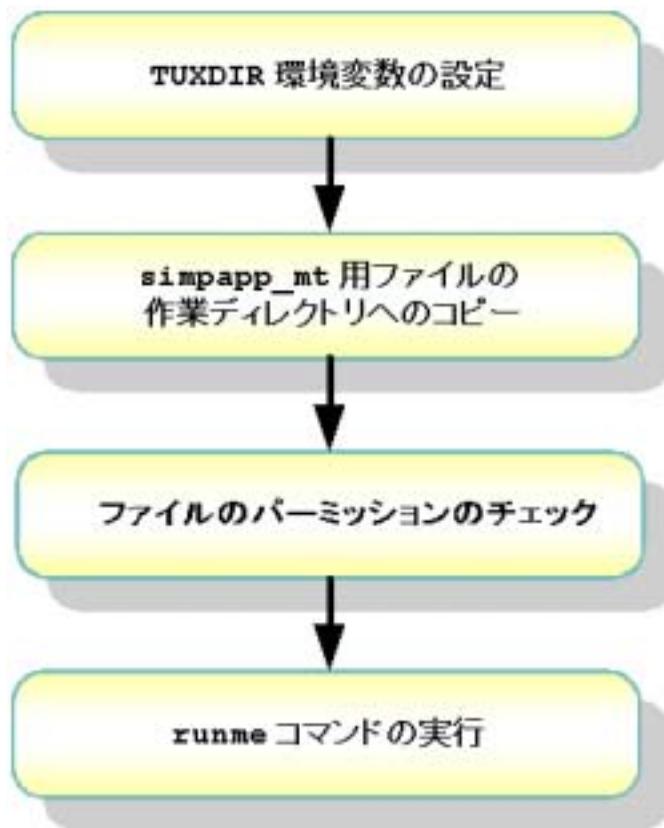
interface SimplePerRequestFactory
{
    Simple find_simple();
};

interface SimplePerObjectFactory
{
    Simple find_simple();
};
```

サンプル・アプリケーションのビルドと実行の方法

この節では、simpapp_mt サンプル・アプリケーションをビルドおよび実行するプロセスをステップごとに説明します。次のフローチャートに、このプロセスの要約を示します。以降の節では、これらのタスクを実行する方法について説明します。

図 4-3simpapp_mt のビルドおよび実行プロセス



TUXDIR 環境変数の設定

simpapp_mt サンプル・アプリケーションをビルドおよび実行する前に、TUXDIR 環境変数がシステムで設定されていることを確認してください。通常、この環境変数はインストール・プロセス中に設定されます。この環境変数に適切なディレクトリが定義されていることを確認する必要があります。

TUXDIR 環境変数は、BEA Tuxedo ソフトウェアがインストールされているディレクトリ・パスに設定されている必要があります。たとえば、次のように入力します。

4 マルチスレッド CORBA サーバ・アプリケーションの作成

Windows

```
TUXDIR=D:\TUXDIR
```

UNIX

```
TUXDIR=/usr/local/TUXDIR
```

TUXDIR 環境変数の検証

アプリケーションを実行する前に、次の手順を実行してこの環境変数に正確な情報が定義されているかどうかを確認してください。

Windows

echo コマンドを実行して TUXDIR の設定を表示します。

```
prompt> echo %TUXDIR%
```

UNIX

1. プロンプトに対して ksh コマンドを実行して、Korn シェルを起動します。
2. printenv コマンドを実行して TUXDIR の設定を表示します。

```
ksh prompt> printenv TUXDIR
```

環境変数の設定の変更

環境変数の設定を変更するには、次の手順を実行します。

Windows

set コマンドを実行して、TUXDIR の新しい値を設定します。

```
prompt> set TUXDIR=directorypath
```

UNIX

1. システム・プロンプトに対して ksh コマンドを実行して、Korn シェルを起動します。
2. ksh プロンプトに対して、export コマンドを入力して TUXDIR 環境変数の値を設定します。

```
ksh prompt> export TUXDIR=directorypath
```

サンプル・アプリケーションの作業ディレクトリの作成

注記 simpapp マルチスレッド・サンプルを実行するときどのようなファイルが新しく作成されるかを確認できるよう、作業ディレクトリを使用することをお勧めします。runme コマンドを実行したら、インストール・ディレクトリ内のファイル・セットと作業ディレクトリ内のファイル・セットを比較してください。

simpapp マルチスレッド・サンプル・アプリケーションの必須ファイルは、次のディレクトリに格納されています。

Windows

```
%TUXDIR%\samples\corba\simpapp_mt
```

UNIX

```
$TUXDIR/samples/corba/simpapp_mt
```

simpapp マルチスレッド・ファイルをすべて格納する作業ディレクトリを作成します。

Windows

Windows エクスプローラを使用して simpapp_mt ディレクトリをコピーするか、次のようにコマンド・プロンプトを使用します。

1. simpapp_mt ファイルのコピー先となる作業ディレクトリを作成します。

```
> mkdir work_directory
```

2. simpapp_mt ファイルを作業ディレクトリにコピーします。

```
> copy %TUXDIR%\samples\corba\simpapp_mt\* work_directory
```

3. 作業ディレクトリに移動します。

```
cd work_directory
```

4. 作業ディレクトリ内のすべてのファイルを表示します。

```
prompt> dir
```

```
makefile.mk                simple_per_object_i.h
makefile.nt                 simple_per_object_server.cpp
Readme.txt                  simple_per_request_i.cpp
runme.cmd                   simple_per_request_i.h
```

4 マルチスレッド CORBA サーバ・アプリケーションの作成

```
runme.ksh                simple_per_request_server.cpp
simple.idl                simple_per_request_server.h
simple_client.cpp         thread_macros.cpp
simple_per_object_i.cpp  thread_macros.h
```

UNIX

ユーザ・インターフェイス・ツールを使用して simpapp_mt ディレクトリのコピーを作成するか、次のようにコマンド・プロンプトを使用します。

1. simpapp_mt ファイルのコピー先となる作業ディレクトリを作成します。

```
> mkdir work_directory
```

2. すべての simpapp_mt ファイルを作業ディレクトリにコピーします。

```
> cp $TUXDIR/samples/corba/simpapp_mt/* work_directory
```

3. 作業ディレクトリに移動します。

```
cd work_directory
```

4. 作業ディレクトリ内のすべてのファイルを表示します。

```
$ ls
```

```
makefile.mk          simple_per_object_i.h
makefile.nt          simple_per_object_server.cpp
Readme.txt           simple_per_request_i.cpp
runme.cmd            simple_per_request_i.h
runme.ksh            simple_per_request_server.cpp
simple.idl            simple_per_request_server.h
simple_client.cpp     thread_macros.cpp
simple_per_object_i.cpp thread_macros.h
```

表 4-1 に、アプリケーションをビルドおよび実行するための simpapp_mt ファイルとその説明を示します。

表 4-1 simpapp_mt ファイル

ファイル	説明
makefile.mk	(UNIX) simpapp_mt サンプル・アプリケーション用の Makefile。このファイルを使用してアプリケーションをビルドします。
makefile.nt	(Windows) simpapp_mt サンプル・アプリケーション用の Makefile。このファイルを使用してアプリケーションをビルドします。

4 マルチスレッド CORBA サーバ・アプリケーションの作成

表 4-1 simpapp_mt ファイル (続き)

ファイル	説明
Readme.txt	simpapp_mt サンプル・アプリケーションのビルドと実行に関する情報を提供する Readme ファイル。
runme.cmd	(Windows) simpapp_mt サンプル・アプリケーションをビルドおよび実行するためのコマンド・ファイル。
runme.ksh	(UNIX) simpapp_mt サンプル・アプリケーションをビルドおよび実行するための Korn シェル・スクリプト。
simple.idl	SimplePerRequestFactory、SimplePerObjectFactory、および Simple インターフェイスを宣言する Object Management Group (OMG) インターフェイス定義言語 (IDL) のコード。
simple_client.cpp	simpapp_mt サンプル・アプリケーション用の CORBA クライアント・プログラム・ソース・コード。
simple_per_object_i.cpp	サーバに組み込まれる Simple サーバントと SimplePerObjectFactory サーバントのインプリメンテーションが含まれるソース・コード。CORBA サーバは、オブジェクトごとのスレッド同時実行手法を使用して起動します。
simple_per_object_i.h	サーバに組み込まれる Simple サーバントと SimplePerObjectFactory サーバントを宣言するためのソース・コード。
simple_per_object_server.cpp	simpapp_mt サンプル・アプリケーション、オブジェクトごとのスレッド同時実行手法用の CORBA サーバ・プログラム・ソース・コード。UBBCONFIG ファイルに CONCURR_STRATEGY = PER_OBJECT を設定します。

表 4-1 simpapp_mt ファイル (続き)

ファイル	説明
simple_per_request_i.cpp	リエントラント・サーバに組み込まれる Simple サーバントと SimplePerRequestFactory サーバントのインプリメンテーションが含まれるソース・コード。リエントラント CORBA サーバは、要求ごとのスレッド同時実行手法を使用して起動します。
simple_per_request_i.h	サーバに組み込まれる Simple サーバントと SimplePerRequestFactory サーバントを宣言するためのソース・コード。
simple_per_request_server.cpp	simpapp_mt サンプル・アプリケーション、要求ごとのスレッド同時実行手法用の CORBA サーバ・プログラム・ソース・コード。UBBCONFIG ファイルに CONCURR_STRATEGY = PER_REQUEST を設定します。
simple_per_request_server.h	simpapp_mt サンプル・アプリケーションのユーザ定義 Server クラスに対して必要な宣言が含まれる bootserverclass.h ファイルの例。
thread_macros.cpp	simpapp_mt サンプル・アプリケーションをサポートするプラットフォーム非依存のスレッド・コンビニエンス・マクロ。
thread_macros.h	スレッド・コンビニエンス・マクロ用のすべてのクラスと変数を宣言するためのソース・コード・ファイル。

すべてのファイルのパーミッションのチェック

simpapp_mt サンプル・アプリケーションをビルドおよび実行するには、作業ディレクトリにコピーしたすべてのファイルに対するユーザ・パーミッションと読み取りパーミッションがなければなりません。パーミッションをチェックし、必要な場合はそれらを変更します。

4 マルチスレッド CORBA サーバ・アプリケーションの作成

注記 make ユーティリティがパスに存在することを確認してください。

Windows

```
> attrib -R /S *.*
```

UNIX

```
> /bin/ksh
```

```
> chmod u+r work_directory/*.*
```

Executing the runme Command

この節では、アプリケーションの実行に必要な手順について説明します。次のように `runme` コマンドを入力します。

Windows

```
> cd work_directory
```

```
> ./runme
```

UNIX

```
> /bin/ksh
```

```
> cd work_directory
```

```
> ./runme.ksh
```

`runme` コマンドは、以下の手順を自動化します。

1. `TUXDIR` 環境変数をチェックします。
2. このアプリケーションで使用される環境変数を設定します。
3. 適切な `bin` ディレクトリが `PATH` にあることを確認します。
4. このスクリプトが実行されるのが初めてではない場合、不要なファイルをディレクトリから削除します。
5. このスクリプトを実行することによって得られる結果を格納するためのディレクトリを作成します。

6. `setenv.ksh` ファイル (UNIX) または `setenv.bat` ファイル (Windows) を作成し、このサンプルをステップ・バイ・ステップ・モードでビルドおよび実行できるようにします。
7. このサンプル用の `ubb` コンフィギュレーション・ファイルを作成します。
8. クライアントのユーザ入力を格納するファイルを作成します。
9. クライアントからの期待される出力を格納するファイルを作成します。
10. サンプルをビルドします。
11. コンフィギュレーション・ファイルをロードします。
12. オブジェクトごとのスレッド・サーバを起動します。
13. 要求ごとのスレッド・サーバを起動します。
14. クライアントを実行し、出力を取得します。
15. その出力と期待された出力を比較します。
16. サーバ・アプリケーションをシャットダウンします。
17. サンプルの実行時に生成されるログを取得します。
18. 結果を保存します。
19. サンプルが正常に実行されたかどうかをユーザに通知します。

`simpapp_mt` サンプル・アプリケーションは、`runme` コマンドの実行中に次のメッセージを出力します。

```
Testing simpapp_mt
  cleaned up
  prepared
  built
  loaded ubb
  booted
  ran
  shutdown
  saved results
PASSED
```

`simpapp_mt` サンプル・アプリケーションのすべての実行時出力は、作業ディレクトリ内の `results` ディレクトリに格納されます。実行時に作成された出力を見るには、次のファイルを参照します。

4 マルチスレッド CORBA サーバ・アプリケーションの作成

- `log` - コンパイル、サーバ起動、またはサーバ・シャットダウンのエラー
- `output` - クライアント・アプリケーションの出力と例外
- `ULOG.date` - サーバ・アプリケーションのエラーと例外

表 4-2 と表 4-3 に、`runme` コマンドの実行によって作成されるファイルとその説明を示します。

表 4-2 作業ディレクトリに作成されるファイル

ファイル	説明
<code>simple_c.cpp</code>	<code>simple.idl</code> ファイルに対する <code>idl</code> コマンドによって作成されます。このモジュールには、 <code>Simple</code> および <code>SimplePerRequestFactory</code> インターフェイス用のクライアント・スタブ機能が含まれます。
<code>simple_c.h</code>	<code>simple.idl</code> ファイルに対する <code>idl</code> コマンドによって作成されます。このモジュールには、 <code>Simple</code> および <code>SimplePerRequestFactory</code> インターフェイスの定義とプロトタイプが含まれます。
<code>simple_s.cpp</code>	<code>simple.idl</code> ファイルに対する <code>idl</code> コマンドによって作成されます。このモジュールには、 <code>Simple_i</code> および <code>SimplePerRequestFactory_i</code> インプリメンテーション用のスケルトン機能が含まれます。
<code>simple_s.h</code>	<code>simple.idl</code> ファイルに対する <code>idl</code> コマンドによって作成されます。このモジュールには、 <code>Simple_i</code> および <code>SimplePerRequestFactory_i</code> インターフェイスの定義とプロトタイプが含まれます。

表 4-2 作業ディレクトリに作成されるファイル (続き)

ファイル	説明
simple_client	simple_c.cpp および simple_client.cpp ファイルに対する buildobjclient コマンドによって作成されます。
simple_per_object_server	simple_c.cpp、simple_s.cpp、simple_per_object_i.cpp、simple_per_object_server.cpp、および thread_macros.cpp ファイルに対する buildobjserver コマンドによって作成されます。
simple_per_request_server	simple_c.cpp、simple_s.cpp、simple_per_request_i.cpp、simple_per_request_server.cpp、および thread_macros.cpp ファイルに対する buildobjserver コマンドによって作成されます。
results directory	このスクリプトの実行結果を取得するために runme コマンドによって作成されます。
adm directory	セキュリティ暗号化キー・データベース・ファイルを格納するために runme コマンドによって作成されます。

表 4-3 results ディレクトリに作成されるファイル

ファイル	説明
input	runme コマンドが C++ クライアント・アプリケーションに提供する入力を格納するために runme コマンドによって作成されます。

4 マルチスレッド CORBA サーバ・アプリケーションの作成

表 4-3 results ディレクトリに作成されるファイル (続き)

ファイル	説明
output	runme コマンドが C++ クライアント・アプリケーションを実行するときの出力を格納するために runme コマンドによって作成されます。
expected_output	runme コマンドが実行されるときに期待される出力を格納するために runme コマンドによって作成されます。この出力ファイルを比較して、テストに合格したかどうかを決定します。
log	runme コマンドによって生成される出力を格納するために runme コマンドによって作成されます。このコマンドが失敗した場合、このファイルと ULOG ファイルでエラーをチェックします。
setenv.cmd	(Windows) simpapp_mt サンプル・アプリケーションをステップ・バイ・ステップ・モードでビルドおよび実行するために必要な環境変数を設定するためのコマンド・ファイル。
setenv.ksh	(UNIX) simpapp_mt サンプル・アプリケーションをステップ・バイ・ステップ・モードでビルドおよび実行するために必要な環境変数を設定するためのコマンド・ファイル。
stderr	tmboot によって生成されるメッセージが含まれます。-noredirect サーバ・オプションが UBBCONFIG ファイルに指定されている場合、fprintf メソッドは出力をこのファイルに送信します。
stdout	tmboot によって生成されるメッセージが含まれます。-noredirect サーバ・オプションが UBBCONFIG ファイルに指定されている場合、fprintf メソッドは出力をこのファイルに送信します。

表 4-3 results ディレクトリに作成されるファイル (続き)

ファイル	説明
tmsysevt.dat	runme コマンドの tmboot コマンドによって生成されます。このファイルには、TMSYSEVT プロセスで使用されるフィルタ規則と通知規則が含まれます。
tuxconfig	コンフィギュレーション・ファイルのバイナリ・バージョン。
ubb	simpapp_mt サンプル・アプリケーション用の UBBCONFIG ファイル。
ULOG.date	実行時エラーを格納するための ULOG ファイル。

サンプル・アプリケーションのステップ・バイ・ステップ実行

この節では、simpapp_mt サンプル・アプリケーションをステップ・バイ・ステップ・モードで実行する方法について説明します。simpapp_mt をステップ・バイ・ステップ・モードで実行するには、あらかじめ runme コマンドを実行しておく必要があります。

次の手順に従って、simpapp_mt アプリケーションを実行します。

1. 環境変数を設定します。

Windows

```
> ..\results\setenv
```

UNIX

```
> ../results/setenv.ksh
```

2. tmboot -y を実行してアプリケーションを起動します。次のような情報が表示されます。

```
>tmboot -y
Booting all admin and server processes in /work_directory/results/tuxconfig
Booting admin processes ...
exec BBL -A : process id=212 ... Started.
```

4 マルチスレッド CORBA サーバ・アプリケーションの作成

Booting server processes ...

```
exec TMSYSEVT -A : process id=289 ... Started.
exec TMFFNAME -A -- -N -M : process id=297 ... Started.
exec TMFFNAME -A -- -N : process id=233 ... Started.
exec TMFFNAME -A -- -F : process id=265 ... Started.
exec simple_per_object_server -A : process id=116 ... Started.
exec simple_per_request_server -A : process id=127 ... Started.
exec ISL -A -- -n //MrBeaver:2468 : process id=270 ... Started.
7 processes started.
>
```

表 4-4 に、`tmboot` によって起動するサーバ・プロセスを示します。

表 4-4 `tmboot` によって起動するサーバ・プロセス

プロセス	説明
TMSYSEVT	システム EventBroker。
TMFFNAME	TMFFNAME サーバ・プロセス。 <ul style="list-style-type: none">■ Master NameManager - <code>-N</code> オプションと <code>-M</code> オプションを指定したときに起動する TMFFNAME サーバ・プロセス。■ SLAVE NameManager - <code>-N</code> オプションだけを指定したときに起動する TMFFNAME サーバ・プロセス。■ FactoryFinder オブジェクト - <code>-F</code> オプションにこのオブジェクトが指定されているときに起動する TMFFNAME サーバ・プロセス。
simple_per_object_server	オブジェクトごとのスレッド・サーバとして起動します。
simple_per_request_server	リエントラントな要求ごとのスレッド・サーバとして起動します。
ISL	IIOP リスナ・プロセス。

3. クライアント・アプリケーションを実行します。

Windows

```
> .\simple_client
```

```
UNIX
```

```
> ./simple_client
```

クライアント・アプリケーションを実行すると、次のようなメッセージが表示されます。

コードリスト 4-3 simpapp_mt クライアントの実行時に表示されるメッセージ

```
Number of simultaneous requests to post (1-50)?
String to convert using thread-per-request server?
Sending 4 deferred forward_lower requests
forward_lower request #0
returned:aabbccddeeffgghhiijjkllmmnnooppqrrssttuuvvwxyzz
forward_lower request #1
returned:aabbccddeeffgghhiijjkllmmnnooppqrrssttuuvvwxyzz
forward_lower request #2
returned:aabbccddeeffgghhiijjkllmmnnooppqrrssttuuvvwxyzz
forward_lower request #3
returned:aabbccddeeffgghhiijjkllmmnnooppqrrssttuuvvwxyzz
Sending 4 deferred forward_upper requests
forward_upper request #0 returned:
AABBCCDDEEFFGGHHIIJJKLLMMNNOOPPQRRSSTTUUVVWXXYYZZ
forward_upper request #1 returned:
AABBCCDDEEFFGGHHIIJJKLLMMNNOOPPQRRSSTTUUVVWXXYYZZ
forward_upper request #2 returned:
AABBCCDDEEFFGGHHIIJJKLLMMNNOOPPQRRSSTTUUVVWXXYYZZ
forward_upper request #3 returned:
AABBCCDDEEFFGGHHIIJJKLLMMNNOOPPQRRSSTTUUVVWXXYYZZ
String to convert using thread-per-object server?
Sending 4 deferred forward_lower requests
forward_lower request #0
returned:aabbccddeeffgghhiijjkllmmnnooppqrrssttuuvvwxyzz
forward_lower request #1
returned:aabbccddeeffgghhiijjkllmmnnooppqrrssttuuvvwxyzz
forward_lower request #2
returned:aabbccddeeffgghhiijjkllmmnnooppqrrssttuuvvwxyzz
forward_lower request #3
returned:aabbccddeeffgghhiijjkllmmnnooppqrrssttuuvvwxyzz
Sending 4 deferred forward_upper requests
forward_upper request #0 returned:
AABBCCDDEEFFGGHHIIJJKLLMMNNOOPPQRRSSTTUUVVWXXYYZZ
forward_upper request #1 returned:
AABBCCDDEEFFGGHHIIJJKLLMMNNOOPPQRRSSTTUUVVWXXYYZZ
```

4 マルチスレッド CORBA サーバ・アプリケーションの作成

```
forward_upper request #2 returned:
AABBCDDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWVWXYZZ

forward_upper request #3 returned:
AABBCDDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWVWXYZZ
```

サンプル・アプリケーションのシャットダウン

別のサンプル・アプリケーションを実行する前に、simpapp_mt サンプル・アプリケーションをシャットダウンして、不要なファイルを作業ディレクトリからすべて削除する必要があります。

1. アプリケーションを終了するには、`tmshutdown -y` コマンドを実行します。次のような情報が表示されます。

```
>tmshutdown -y
Shutting down all admin and server processes in /work_directory/results/tuxconfig

Shutting down server processes ...

Server Id = 5 Group Id = SYS_GRP Machine = SITE1:      shutdown succeeded.
Server Id = 2 Group Id = APP_GRP2 Machine = SITE1:    shutdown succeeded.
Server Id = 4 Group Id = SYS_GRP Machine = SITE1:    shutdown succeeded.
Server Id = 3 Group Id = SYS_GRP Machine = SITE1:    shutdown succeeded.
Server Id = 2 Group Id = SYS_GRP Machine = SITE1:    shutdown succeeded.
Server Id = 1 Group Id = SYS_GRP Machine = SITE1:    shutdown succeeded.

Shutting down admin processes ...

Server Id = 0 Group Id = SITE1 Machine = SITE1: shutdown succeeded.
7 processes stopped.
```

2. 作業ディレクトリを元の状態に戻します。

Windows

```
> ..\results\setenv
> make -f clean
```

UNIX

```
> ../results/setenv.ksh
> make -f makefile.mk clean
```

マルチスレッド CORBA サーバ・アプリケーションの管理

ここでは、以下の内容について説明します。

- スレッド・プール・サイズの指定
- スレッド・モデルの指定
- 活性化されたオブジェクトの数の指定
- UBBCONFIG ファイルの例

スレッド・プール・サイズの指定

スレッド・プールの最小サイズと最大サイズを指定するための `MAXDISPATCHTHREADS` パラメータと `MINDISPATCHTHREADS` パラメータは、`UBBCONFIG` ファイルの `SERVERS` セクションに存在します。これらのパラメータの指定例については、リスト 4-4 を参照してください。マルチスレッド CORBA アプリケーションは、これらの値を使用してスレッド・プールを作成および管理します。

MAXDISPATCHTHREADS

`MAXDISPATCHTHREADS` パラメータは、各サーバ・プロセスで生成される、同時に実行できるディスパッチ・スレッドの最大数を決定します。このパラメータを指定する際には、次の事項を考慮してください。

- `MAXDISPATCHTHREADS` の値によって、受信する要求を格納するために拡大できる最大サイズが決定されます。
- `MAXDISPATCHTHREADS` のデフォルト値は 1 です。1 より大きい値を指定した場合、特別なディスパッチ・スレッドが作成および使用されます。このディスパッチャ・スレッドは、スレッド・プールの最大サイズを決定するスレッド数には、含まれていません。

4 マルチスレッド CORBA サーバ・アプリケーションの作成

注記 MAXDISPATCHTHREADS に 1 より大きい値を指定し、CONCURR_STRATEGY スレッド・モデル・パラメータの値を指定しなかった場合、アプリケーションのスレッド・モデルはデフォルトによってオブジェクトごとのスレッドに設定されます。CONCURR_STRATEGY スレッド・モデル・パラメータについては、「第 4 章の -65 ページ「スレッド・モデルの指定」」を参照してください。

- MAXDISPATCHTHREADS パラメータの値を 1 に設定した場合、CORBA サーバ・アプリケーションをシングル・スレッド・サーバとして構成する必要があります。

注記 buildobjserver -t を指定してマルチスレッド CORBA サーバ・アプリケーションをビルドした場合、そのサーバはマルチスレッド・モードで動作できます。マルチスレッド CORBA サーバ・アプリケーションとして実行するには、UBBCONFIG ファイルの MAXDISPATCHTHREADS パラメータを 1 より大きい値に設定する必要があります。それ以外の値を設定した場合、サーバ・アプリケーションはシングル・スレッド・モードで動作します。

- MAXDISPATCHTHREADS パラメータに指定する値は、MINDISPATCHTHREADS パラメータに指定する値を下回ってはいけません。
- オペレーティング・システムにより、1 つのプロセスで作成できるスレッドの最大数は制限されます。MAXDISPATCHTHREADS は、その制限値より少ない、アプリケーションが必要とするアプリケーション管理スレッドの数を差し引いた値にします。

MAXDISPATCHTHREADS パラメータの値は、ほかのパラメータにも影響を与えます。たとえば、MAXACCESSORS パラメータは BEA Tuxedo システムへの同時アクセス数を制御します、各スレッドは、1 つのアクセサとしてカウントされます。マルチスレッド・サーバのアプリケーションの場合、各サーバの実行がコンフィギュレーションされているシステム管理のスレッドの数を考慮しておく必要があります。A システム管理のスレッドとは、アプリケーションで開始され管理されるスレッドに対して、BEA Tuxedo ソフトウェアで開始され管理されるスレッドです。内部では BEA Tuxedo が、利用可能なシステム管理のスレッドのプールを管理しています。クライアント要求が受信されると、スレッド・プールから利用可能なシステム管理のスレッドがその要求を実行するように、スケジューリングされています。要求が完了すると、システム管理のスレッドは利用可能なスレッドのプールに返されます。

たとえば、システム内に 4 つのマルチスレッド・サーバがあり、各サーバが 50 個のシステム管理のスレッドを実行するようにコンフィギュレーションされている場合、これらのサーバにおけるアクセサ要件は、次のように計算される、アクセサ数の合計となります。

$$50 + 50 + 50 + 50 = 200 \text{ accessors}$$

MINDISPATCHTHREADS

サーバが最初にブートされたときに開始されるサーバ・ディスパッチ・スレッドの数を指定するには、MINDISPATCHTHREADS パラメータを使用します。このパラメータを指定する際には、次の事項を考慮してください。

- MINDISPATCHTHREADS の値によって、スレッド・プール内のスレッドの初期割り当てが決定されます。
- MAXDISPATCHTHREADS が 1 より大きい場合に作成される別個のディスパッチャ・スレッドは、MINDISPATCHTHREADS の範囲には含まれません。
- MINDISPATCHTHREADS に指定する値は、MAXDISPATCHTHREADS に指定する値を上回ってはいけません。
- MINDISPATCHTHREADS のデフォルト値は 0 です。

スレッド・モデルの指定

スレッド・モデルを指定するには、UBBCONFIG ファイルの SERVERS セクションに定義されている CONCURR_STRATEGY パラメータを設定します。

CONCURR_STRATEGY パラメータを使用して、マルチスレッド CORBA サーバ・アプリケーションが使用するスレッド・モデルを指定します。CONCURR_STRATEGY パラメータは、次のいずれかの値を受け付けます。

- CONCURR_STRATEGY = PER_REQUEST
- CONCURR_STRATEGY = PER_OBJECT

CONCURR_STRATEGY = PER_REQUEST を指定して要求ごとのスレッド・モデルを採用した場合、CORBA サーバ・アプリケーションの各呼び出しはスレッド・プール内の任意のスレッドに割り当てられます。

CONCURR_STRATEGY = PER_OBJECT を指定してオブジェクトごとのスレッド・モデルを採用した場合、活性化された各オブジェクトは常に1つのスレッドに関連付けられます。オブジェクトに対する要求ごとに、ディスパッチ・スレッドとオブジェクトとの関連付けが確立されます。

MAXDISPATCHTHREADS の値が1より大きく、CONCURR_STRATEGY の値を指定しなかった場合、スレッド・モデルは PER_OBJECT に設定されます。

スレッド・モデルの特性については、「第4章の28ページ「スレッド・モデル」」を参照してください。

活性化されたオブジェクトの数の指定

掲示板のアクティブ・オブジェクト・マップ表に格納されるマシンごとのオブジェクトの最大数を指定するには、MAXOBJECTS パラメータを使用します。この値は、コンフィギュレーション・ファイルの RESOURCES セクションか MACHINES セクションのいずれかで設定できます。RESOURCES セクションの MAXOBJECTS number は、システム全体にわたる設定です。システム全体にわたる設定をマシンごとに上書きするには、MACHINES セクションの MAXOBJECTS number を使用します。

システム全体にわたる設定の場合、次のように指定します。

```
*RESOURCES  
    MAXOBJECTS number
```

特定のマシンのシステム全体にわたる設定を上書きするには、次のように指定します。

```
*MACHINES  
    MAXOBJECTS = number
```

number の値は、オペレーティング・システムのリソースによってのみ制限されます。

UBBCONFIG ファイルの例

リスト 4-4 に、BEA Tuxedo Threads サンプル・アプリケーション用の UBBCONFIG ファイルを示します。スレッドに関連するパラメータは、太字で示してあります。

注記 **MAXOBJECTS** パラメータの値は、マルチスレッド・サーバのオペレーションに影響を与えます。ただし、このパラメータはマルチスレッド・サーバに固有のものではありません。この値は、シングル・スレッド・サーバのオペレーションにも影響を与えます。**MAXOBJECTS** の値を大きくすると、サーバで消費されるシステム・リソースが増加します。

コード リスト 4-4 Threads サンプル・アプリケーションの UBBCONFIG ファイル

```
*RESOURCES
  IPCKEY      55432
  DOMAINID   simpapp
MAXOBJECTS 100
  MASTER     SITE1
  MODEL      SHM
  LDBAL      N

*MACHINES
  "sunstar"
  LMID       = SITE1
  APPDIR     = "/rusers1/lyon/samples/corba/simpapp_mt"
  TUXCONFIG  = "/rusers1/lyon/samples/corba/simpapp_mt/results/tuxconfig"
  TUXDIR     = "/usr/local/TUXDIR"
  MAXWSCLIENTS = 10
MAXACCESSERS = 200

*GROUPS
  SYS_GRP
    LMID     = SITE1
    GRPNO   = 1
  APP_GRP1
    LMID     = SITE1
    GRPNO   = 2
  APP_GRP2
```

4 マルチスレッド CORBA サーバ・アプリケーションの作成

```
LMID      = SITE1
GRPNO     = 3

*SERVERS
  DEFAULT:
    RESTART = Y
    MAXGEN  = 5
  TMSYSEVT
    SRVGRP  = SYS_GRP
    SRVID   = 1
  TMMFFNAME
    SRVGRP  = SYS_GRP
    SRVID   = 2
    CLOPT   = "-A -- -N -M"
  TMMFFNAME
    SRVGRP  = SYS_GRP
    SRVID   = 3
    CLOPT   = "-A -- -N"
  TMMFFNAME
    SRVGRP  = SYS_GRP
    SRVID   = 4
    CLOPT   = "-A -- -F"
  simple_per_object_server
    SRVGRP  = APP_GRP1
    SRVID   = 1
    MINDISPATCHTHREADS = 10
    MAXDISPATCHTHREADS = 100
    CONCURR_STRATEGY = PER_OBJECT
    RESTART = N
  simple_per_request_server
    SRVGRP  = APP_GRP2
    SRVID   = 2
    MINDISPATCHTHREADS = 10
    MAXDISPATCHTHREADS = 100
    CONCURR_STRATEGY = PER_REQUEST
    RESTART = N
  ISL
    SRVGRP  = SYS_GRP
    SRVID   = 5
    CLOPT   = "-A -- -n //sunbstar:2468 -d /dev/tcp"

*SERVICES
```

5 セキュリティと CORBA サーバ・ア プリケーション

この章では、セキュリティと CORBA サーバ・アプリケーションについて、Security University サンプル・アプリケーションを例にして説明します。この Security サンプル・アプリケーションでインプリメントするセキュリティ・モデルでは、University サンプル・アプリケーションの学生ユーザはそのアプリケーションへのログイン・プロセスの一部として認証を受ける必要があります。

ここでは、以下の内容について説明します。

- セキュリティと CORBA サーバ・アプリケーションの概要
- University サーバ・アプリケーションの設計上の考慮事項

セキュリティと CORBA サーバ・アプリケーションの概要

一般に、CORBA サーバ・アプリケーションはセキュリティにほとんど関係しません。BEA Tuxedo ドメインでのセキュリティはシステム管理者によって `UBBCONFIG` ファイルで指定され、ドメインへのログインや認証はクライアント・アプリケーションによって処理されます。BEA Tuxedo システムでサポートされているセキュリティ・モデルのいずれにも、BEA Tuxedo ドメインで稼働するサーバ・アプリケーションに関する要件はありません。

しかし、使用する CORBA アプリケーションのセキュリティ・モデルをインプリメントまたは改善する際に、オブジェクトを追加したり、既存のオブジェクトにオペレーションを追加したりすることになり、そうして追加されたオブジェクトやオペレーションがサーバ・アプリケーションの管理対象である場合もあります。

この章では、University サーバ・アプリケーションを拡張して学生という概念を追加する方法を説明します。この拡張はクライアント・アプリケーションに組み込まれて、クライアント・アプリケーションのユーザの識別、およびログインの手段となります。

クライアント・アプリケーションで BEA Tuxedo ドメインへの認証を受ける方法の詳細については、『BEA Tuxedo CORBA クライアント・アプリケーションの開発方法』を参照してください。BEA Tuxedo ドメインにセキュリティ・モデルをインプリメントする方法の詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。

University サーバ・アプリケーションの設計上の考慮事項

Security University サンプル・アプリケーションの設計の要点は、クライアント・アプリケーションのユーザに対してログオンを求め、ログオンしないと何もできないようにすることです。このため、Security サンプル・アプリケーションでは、ユーザの概念を定義する必要があります。

アプリケーションにログオンするために、クライアント・アプリケーションは BEA Tuxedo ドメインのセキュリティ・サービスに以下の情報を提供する必要があります (このアプリケーションの学生ユーザは、ユーザ名とアプリケーション・パスワードのみを提供します)。

- クライアント名
- ユーザ名
- アプリケーション・パスワード

Security サンプル・アプリケーションは、`Registrar` オブジェクトに `get_student_details()` オペレーションを追加します。このオペレーションによって、クライアント・アプリケーションは、BEA Tuxedo ドメインにログオンした後で University データベースから学生の情報を取得することができます。

注記 `get_student_details()` オペレーションは、BEA Tuxedo ドメインのセキュリティ・モデルをインプリメントすることとは無関係です。このオペレーションを追加しても、Security サンプル・アプリケーションに補足機能を追加するのみです。Security サンプル・アプリケーションに追加されるセキュリティ・モデルの詳細、およびクライアント・アプリケーションが Security サーバ・アプリケーションにログオンする方法の詳細については、『BEA Tuxedo CORBA クライアント・アプリケーションの開発方法』を参照してください。

ここでは、次の内容について説明します。

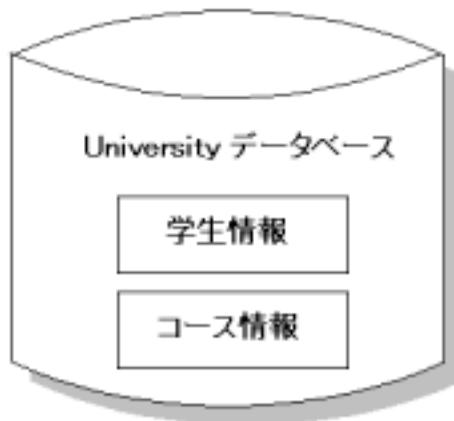
- Security University サンプル・アプリケーションのしくみ

- クライアント・アプリケーションに学生の詳細情報を返す機能を設計する際の考慮事項

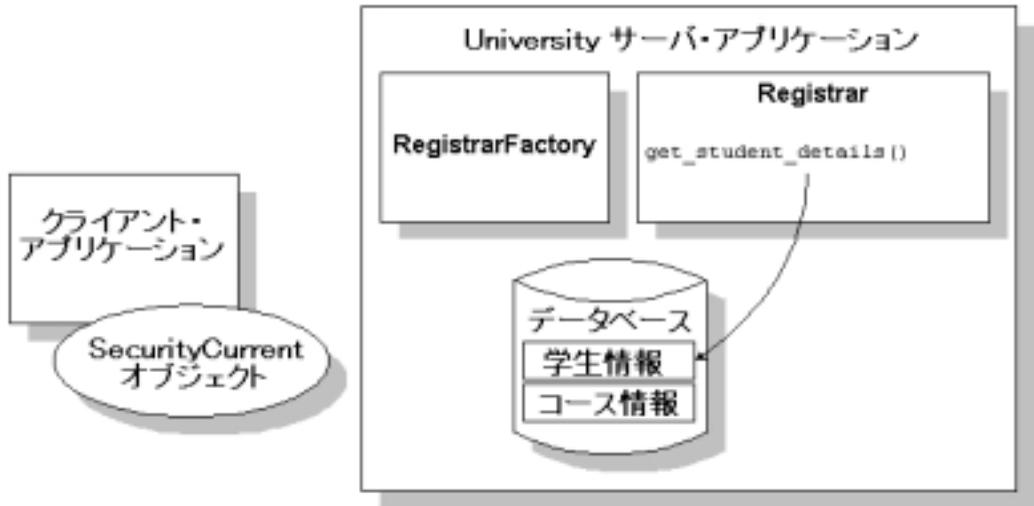
Security University サンプル・アプリケーションのしくみ

Security サンプル・アプリケーションをインプリメントするために、クライアント・アプリケーションではログオンについて学生エンド・ユーザとのやり取りが追加されます。このやり取りではクライアント・マシン上のローカルな SecurityCurrent オブジェクトを使用して PrincipalAuthenticator オブジェクトのオペレーションが呼び出され、BEA Tuxedo ドメインにアクセスするためのログイン手順の一部となります。ユーザ認証のプロセスの後、クライアント・アプリケーションは Registrar オブジェクトの `get_student_details()` を呼び出して、学生ユーザについての情報を取得します。

Security サンプル・アプリケーションで使用される University データベースは、コース情報に加えて学生の情報も含むように更新されて、次の図のようになります。



`get_student_details()` オペレーションはデータベースで学生情報の部分にアクセスして、クライアントのログオン・オペレーションに必要な学生情報を取得します。次の図は、Security サンプル・アプリケーションに関する主要なオブジェクトを示します。



Security サンプル・アプリケーションの一般的な使用のシナリオには、以下のようなイベントのシーケンスが含まれることがあります。

1. クライアント・アプリケーションは、SecurityCurrent オブジェクトへのリファレンスを Bootstrap オブジェクトから取得します。
2. クライアント・アプリケーションは SecurityCurrent オブジェクトを呼び出して、BEA Tuxedo ドメインで要求されるセキュリティのレベルを判別します。
3. クライアント・アプリケーションは学生ユーザについて学生 ID および必要なパスワードを問い合わせます。
4. クライアント・アプリケーションは、学生についての情報を認証サービスから取得して、学生の認証を行います。
5. 認証の処理が正常に終了したら、クライアント・アプリケーションは BEA Tuxedo ドメインにログオンします。

6. クライアント・アプリケーションは Registrar オブジェクトの `get_student_details()` オペレーションを呼び出し、学生 ID を渡して、学生についての情報を取得します。
7. Registrar オブジェクトはデータベースをスキャンして、クライアント要求にある学生 ID と一致する学生情報を探します。
8. クライアント・アプリケーションの要求で指定されている学生 ID と、データベースにある学生情報の間に一致があれば、Registrar オブジェクトはクライアント・アプリケーションに `struct StudentDetails` を返します。学生によって入力された ID がデータベースにある情報と一致しない場合、Registrar オブジェクトはクライアント・アプリケーションに CORBA 例外を返します。
9. Registrar オブジェクトがクライアント・アプリケーションに `StudentDetails` を返した場合、クライアント・アプリケーションは学生ユーザ個人用の許可メッセージを表示します。

クライアント・アプリケーションに学生の詳細情報を返す機能を設計する際の考慮事項

クライアント・アプリケーションは、ユーザが University アプリケーションの使用を継続できるように、BEA Tuxedo システムにユーザを記録する方法を提供する必要があります。このためには、クライアント・アプリケーションでユーザを識別する手段が必要です。Security サンプル・アプリケーションでは、学生 ID で識別を行います。

University サーバ・アプリケーションに必要な処理は、クライアント・アプリケーションがユーザ認証プロセスを完了できるように、学生 ID に基づいて学生のデータを返すことのみです。このため、Security サンプル・アプリケーションの OMG IDL によって、Registrar オブジェクトに `get_student_details()` オペレーションの定義が追加されています。University サーバ・アプリケーションの設計上の主な考慮事項は、既に説明したとおりオペレーション上のシナリオに基づいています。つまり、1名の学生が一度に1つのクライアント・アプリケーションとやり取りできるよう

にすることです。このため、サーバ・アプリケーションは `get_student_details()` オペレーションのインプリメンテーションで大量のデータ処理を考慮する必要がありません。

`get_student_details()` オペレーションの OMG IDL 定義は次のとおりです。

```
struct StudentDetails
{
    StudentId      student_id;
    string         name;
    CourseDetailsList registered_courses;
};
```


6 トランザクションの CORBA サーバ・ア プリケーションへの 統合

この章では、CORBA サーバ・アプリケーションにトランザクションを統合する方法について、Transactions University サンプル・アプリケーションを例にして説明します。Transactions サンプル・アプリケーションは、学生がコースのセットを登録するプロセスをカプセル化します。Transactions サンプル・アプリケーションでは、CORBA サーバ・アプリケーションをトランザクションに統合するためのすべての方法ではなく、トランザクションの振る舞いの2つのモデルを示すことによって、アプリケーション一般、特にオブジェクトの永続状態に対するトランザクションの振る舞いの影響を示します。

ここでは、以下の内容について説明します。

- BEA Tuxedo システムでのトランザクションの概要
- CORBA サーバ・アプリケーションでのトランザクションの設計およびインプリメンテーション
- トランザクションの CORBA クライアントおよびサーバ・アプリケーションへの統合。この節では、次の内容について説明します。

- オブジェクトを自動的にトランザクションに關与させる方法
 - オブジェクトのトランザクションへの参加の有効化
 - トランザクションのスコープ指定時のオブジェクト呼び出しの防止
 - 実行中のトランザクションからのオブジェクトの除外
 - 方針の割り当て
 - XA リソース・マネージャのオープン
 - XA リソース・マネージャのクローズ
- トランザクション管理とオブジェクト状態管理
 - BEA Tuxedo のトランザクションの使用に関する注意事項
 - ユーザ定義の例外

また、この章ではユーザ定義例外についても説明します。Transactions サンプル・アプリケーションはユーザ定義例外を利用します。この例外は、クライアント・アプリケーションに返すことができ、クライアントが開始したトランザクションのロールバックを発生させることが可能です。

BEA Tuxedo システムでのトランザクションの概要

BEA Tuxedo システムでは、データベースのトランザクションが正確に完了すること、およびデータベースのトランザクションが高性能なトランザクションの ACID プロパティ（原子性、一貫性、独立性、および持続性）のすべてを備えることを保証する手段として、トランザクションが提供されています。つまり、永続ストレージに複数の書き込みオペレーションを実行する上での要件があるので、オペレーションの成功が保証されている必要があります。オペレーションが1つでも失敗すれば、一連のオペレーションの全体がロールバックされます。

一般に、トランザクションは次のリストで説明される状況に適しています。それぞれの状況は、BEA Tuxedo システムでサポートされているトランザクション・モデルをカプセル化しています。

- クライアント・アプリケーションは複数の異なるオブジェクトへの呼び出しを行う必要があり、このために1つまたは複数のデータベースへの書き込みオペレーションが関係します。呼び出しの1つが失敗した場合は、(メモリ、またはより一般的にはデータベースに)書き込まれた状態をロールバックする必要があります。

たとえば、旅行代理店アプリケーションがあるとします。クライアント・アプリケーションは、たとえばフランスのストラスプールからオーストラリアのアリス・スプリングスまでなど、遠隔地への旅行を手配する必要があります。このような旅行では、複数の予約が必要になります。クライアント・アプリケーションでは、旅程の各区分の予約が順番に行われます。たとえば、ストラスプールからパリ、パリからニューヨーク、ニューヨークからロサンゼルスへの予約が順番に行われます。しかし、いずれかのフライトが予約できないとき、クライアント・アプリケーションにはその時点で予約済みのほかのフライトをすべてキャンセルする方法が必要です。たとえば、クライアント・アプリケーションでロサンゼルスからホノルルへの指定日のフライトを予約できなかったら、クライアント・アプリケーションはその時点までに済ませたフライト予約をキャンセルする必要があります。

- クライアントには、サーバ・アプリケーションによって管理されているオブジェクトと会話して、特定のオブジェクトのインスタンスに対する複数の呼び出しを実行する必要があります。会話処理には、以下のような1つまたは複数の特徴があります。
 - データは、それぞれの呼び出しの最中またはその後メモリにキャッシュされるか、またはデータベースに書き込まれます。
 - データは、会話の最後にデータベースに書き込まれます。
 - クライアントには、各呼び出し間のメモリ内のコンテキストを保持するオブジェクトが必要です。つまり、連続した呼び出しの各回では、会話の間にメモリに保持されているデータが使用されます。
 - 会話の終了時には、会話の間または終了時に発生したデータベース書き込みオペレーションのすべてをクライアントがキャンセルできる必要があります。

たとえば、インターネット・ベースのオンライン・ショッピング・アプリケーションを検討してみます。クライアント・アプリケーションのユーザがオンライン・カタログを一覧して、複数の購入を選択します。ユーザは、購入したい商品をすべて選択したら、ボタンをクリックして購入を実行し、クレジット・カード情報を入力します。クレジット・カードの確認が失敗したら（たとえば、ユーザのクレジット・カード情報が無効だった場合）、ショッピング・アプリケーションでは、未決済の購入選択をすべてキャンセルしたり、会話で行われたすべての購入トランザクションをロールバックしたりする必要があります。

- オブジェクトに対する単一のクライアント呼び出しのスコープ内で、オブジェクトは、データベース内のデータに対する複数の編集を実行します。編集のいずれかが失敗した場合、オブジェクトはすべての編集をロールバックするメカニズムを必要とします。そして、この状況では、個々のデータベース編集は必ずしも CORBA 呼び出しである必要がありません。

たとえば、銀行取引アプリケーションがあるとします。クライアントは、窓口オブジェクトに対して振替オペレーションを呼び出します。振替オペレーションは、銀行データベースに対して以下の呼び出しを実行するために窓口オブジェクトを必要とします。

- ある口座に対する振替元メソッドの呼び出し
- 別の口座に対する振替先メソッドの呼び出し

銀行データベースの振替先呼び出しに失敗した場合、銀行取引アプリケーションではそれまでの振替元呼び出しをロールバックする機能が必要です。

CORBA サーバ・アプリケーションでのトランザクションの設計およびインプリメンテーション

ここでは、CORBA サーバ・アプリケーションでのトランザクションを設計およびインプリメントする方法について、Transactions University サンプル・アプリケーションを例にして説明します。また、ここでは、Transactions サンプル・アプリケーションの動作と、トランザクションをインプリメントする際の設計上の考慮事項についても説明します。トランザクション全般の詳細については、第 6 章の 12 ページ「トランザクションの CORBA クライアントおよびサーバ・アプリケーションへの統合」を参照してください。

Transactions サンプル・アプリケーションは、学生がコースのセットを登録するプロセスを、トランザクションを利用してカプセル化します。このアプリケーションで使用されているトランザクション・モデルは、前の節で説明したように、会話モデルと、単一の呼び出しがデータベースに対して複数のオペレーションを個別に行うモデルを組み合わせたものです。

この Transactions サンプル・アプリケーションは、Security サンプル・アプリケーションを基に以下の機能を追加したものです。

- 学生は、登録したいコースのリストを提出できます。各コースは番号で示されます。
- リストの各コースについて、University サーバ・アプリケーションは次の項目を確認します。
 - コースが University データベースにあるかどうか。
 - 学生が既に登録されているかどうか。
 - 学生が履修できる単位の最大数を超えているかどうか。
- コースについて確認した結果に問題がなければ、University サーバ・アプリケーションは学生をコースに登録します。

6 トランザクションの CORBA サーバ・アプリケーションへの統合

- コースがデータベースに存在しない、または当該の学生が既にコースに登録済みであるといった理由でサーバ・アプリケーションがコースに学生を登録できない場合、サーバ・アプリケーションはクライアント・アプリケーションに対して、登録のプロセスに失敗したコースのリストを返します。このとき、クライアント・アプリケーションは、登録のプロセスが成功したコースについて学生の登録のトランザクションをコミットするか、トランザクション全体をロールバックするかを選択できます。
- 学生が登録可能なコース数の上限を超えたためにコース登録が失敗した場合、サーバ・アプリケーションはクライアント・アプリケーションに CORBA 例外を返し、コースの登録が失敗した理由を示す短いメッセージを提供します。サーバ・アプリケーションは、当該のトランザクションをロールバック・オンリーとしてマークしません。

Transactions サンプル・アプリケーションは、次の 2 とおりの方法でトランザクションをロールバックします。

- 致命的ではない場合。コースがデータベースに存在しないか、学生が既に登録されているためにコースの登録が失敗する場合は、それらのコースの数がサーバ・アプリケーションからクライアント・アプリケーションに返されます。トランザクションをロールバックするかどうかは、クライアント・アプリケーションのユーザに任せられます (Transaction クライアント・アプリケーション・コードでは、自動的にトランザクションをロールバックします)。
- 致命的な場合。登録できる単位の最大数を超えているためにコースの登録が失敗する場合は、サーバ・アプリケーションによって CORBA 例外が生成され、その例外がクライアントに返されます。この場合も、トランザクションをロールバックするかどうかは、クライアント・アプリケーションに依存します。

したがって、Transactions サンプル・アプリケーションでも、ユーザ定義の CORBA 例外をインプリメントする方法が示されています。たとえば、学生が登録可能なコースの最大数を超えているコースに登録しようとすると、サーバ・アプリケーションは、TooManyCredits 例外を返します。クライアント・アプリケーションは、この例外を受け取ると、自動的にトランザクションをロールバックします。

ここでは、次の内容について説明します。

- Transactions University サンプル・アプリケーションのしくみ

- Transactions University サンプル・アプリケーションで使用するトランザクション・モデル
- University サーバ・アプリケーションのオブジェクト状態に関する注意事項
- Transactions サンプル・アプリケーションのコンフィギュレーションの要件

Transactions University サンプル・アプリケーションのしくみ

学生を登録するプロセスをインプリメントするために、Transactions サンプル・アプリケーションは、以下の作業を実行します。

- クライアント・アプリケーションは、Bootstrap オブジェクトから TransactionCurrent オブジェクトへのリファレンスを取得します。
- 学生が登録希望のコースのリストを提出したときに、クライアント・アプリケーションは以下を実行します。
 - a. TransactionCurrent オブジェクトの `Current::begin()` オペレーションを呼び出して、トランザクションを開始します。
 - b. Registrar オブジェクトの `register_for_courses()` オペレーションを呼び出して、コースのリストを渡します。
- Registrar オブジェクトの `register_for_courses()` オペレーションは、登録の要求を処理するためにループを実行し、リストの各コースについて次の処理を繰り返します。
 - a. 既に学生が登録されているコースの数を調べます。
 - b. 学生が登録済みであるコースを、コースのリストに追加します。Registrar オブジェクトは、発生する可能性のある次の問題を確認します。これらは、トランザクションのコミットを妨げる問題です。
 - 学生がそのコースを既に登録しているかどうか。
 - リスト内にコースが存在しているかどうか。

- 学生が履修できる単位の最大数を超過しているかどうか。
- アプリケーションの OMG IDL での定義に従って、`register_for_courses()` オペレーションは、クライアント・アプリケーションにパラメータ `NotRegisteredList` を返します。このパラメータには、登録に失敗したコースのリストが含まれています。

`NotRegisteredList` の値が空の場合、クライアント・アプリケーションはトランザクションをコミットします。

`NotRegisteredList` 値にコースが含まれていない場合、クライアント・アプリケーションは、登録に成功したコースに対する登録プロセスを完了するかどうかを指定するよう学生に要求します。登録を完了するように選択した場合、クライアント・アプリケーションはトランザクションをコミットします。登録を取り消すように選択した場合、クライアント・アプリケーションはトランザクションをロールバックします。
- 学生が登録の最大数を越えたためにコースの登録が失敗した場合、`Registrar` オブジェクトは `TooManyCredits` 例外をクライアント・アプリケーションに返し、クライアント・アプリケーションはトランザクションの全体をロールバックします。

Transactions University サンプル・アプリケーションで使用するトランザクション・モデル

Transactions サンプル・アプリケーションの基本設計原理は、コース登録を一度に 1 つではなくグループ単位で処理することです。この設計は、`Registrar` オブジェクトへのリモート呼び出しの数を最小にすることに役立ちます。

この設計のインプリメンテーションで、Transactions サンプル・アプリケーションはトランザクションの使用の 1 モデルを示します。トランザクションの各モデルの詳細については、第 6 章の 2 ページ「BEA Tuxedo システムでのトランザクションの概要」を参照してください。このモデルの特徴は、次のとおりです。

- クライアントは、TransactionCurrent オブジェクト上で `begin()` オペレーションを呼び出してトランザクションを開始し、次に Registrar オブジェクト上で `register_for_courses()` オペレーションを呼び出します。

Registrar オブジェクトは、登録可能なコースについて学生を登録してから、登録のプロセスに失敗したコースのリストを返します。クライアント・アプリケーションは、トランザクションをコミットするかロールバックするかを選択できます。トランザクションは、クライアント・アプリケーションとサーバ・アプリケーションの間の会話をカプセル化します。

- `register_for_courses()` オペレーションは、University データベースの複数チェックを実行します。いずれか1つのチェックが失敗した場合、トランザクションをロールバックできます。

University サーバ・アプリケーションのオブジェクト状態に関する注意事項

Transactions University サンプル・アプリケーションはトランザクションに関与するため、University サーバ・アプリケーションは通常、オブジェクト状態に関していくつかの点（特にロールバック時の）を考慮する必要があります。ロールバックが発生した場合、サーバ・アプリケーションは、関連するすべてのオブジェクトが、正しい状態に復元された永続状態を持つことを保証する必要があります。

Registrar オブジェクトがデータベースのトランザクションに使用されるので、このオブジェクトについてはトランザクションに関与させるのが正しい設計上の選択です。つまり、このオブジェクトのインターフェイスに `always` トランザクション方針を割り当てることです。オブジェクト呼び出し時にトランザクションがまだスコープ指定されていない場合、BEA Tuxedo システムは、トランザクションを自動的に開始します。

Registrar オブジェクトを自動的にトランザクション・モードになるようにすることで、このオブジェクトによって実行されるすべてのデータベース書き込みオペレーションは、常にトランザクションのスコープ内で行われることになり、クライアント・アプリケーションによってトランザクションが開始されたかどうかは関係がなくなります。サーバ・アプリケーションは XA

6 トランザクションの CORBA サーバ・アプリケーションへの統合

リソース・マネージャを使用し、またオブジェクトは、データベースに書き込みを実行するときにトランザクションにあることが保証されます。したがって、XA リソース・マネージャがオブジェクトの代わりにロールバックまたはコミットの権限を持つことになるので、オブジェクトにはロールバックまたはコミット権限がありません。

しかし、`RegisterFactory` オブジェクトは、トランザクションの過程で使用されるデータを管理しないのでトランザクションから除外してもかまいません。オブジェクトをトランザクションから除外することで、トランザクションに課せられる処理のオーバーヘッドを最小化します。

Registrar オブジェクトで定義されたオブジェクト方針

`Registrar` オブジェクトをトランザクションに關与するようにするために、ICF ファイルは `Registrar` インターフェイスに `always` トランザクション方針を指定します。したがって、Transactions サンプル・アプリケーションでは、ICF ファイルで、`Registrar` インターフェイスに対して以下のオブジェクト方針を指定します。

活性化方針	トランザクション方針
<code>process</code>	<code>always</code>

RegistrarFactory オブジェクトで定義されたオブジェクト方針

`RegistrarFactory` オブジェクトをトランザクションから除外するために、ICF ファイルは `Registrar` インターフェイスに `ignore` トランザクション方針を指定します。したがって、Transactions サンプル・アプリケーションでは、ICF ファイルで、`RegistrarFactory` インターフェイスに対して以下のオブジェクト方針を指定します。

活性化方針	トランザクション方針
<code>process</code>	<code>ignore</code>

Transactions サンプル・アプリケーションでの XA リソース・マネージャの使用

Transactions サンプル・アプリケーションは、オブジェクト状態データを自動的に処理する Oracle トランザクション・マネージャ・サーバ (TMS) を使用します。XA リソース・マネージャを使用すると、サーバ・アプリケーションで管理される別のオブジェクトが、データベースとの間のデータの読み書きをどのように実行するかについて、以下のような特定の要件が適用されます。

- XA リソース・マネージャの一部 (たとえば、Oracle) では、すべてのデータベース・オペレーションがトランザクションのスコープ内であることが要求されます。これにより、`CourseSynopsisEnumerator` オブジェクトは、データベースからの読み取りを行うので、トランザクション内にスコープ指定される必要があります。
- トランザクションをコミットまたはロールバックすると、XA リソース・マネージャは、そのコミットまたはロールバックが示す永続的な状態を自動的に処理します。つまり、トランザクションをコミットすると、XA リソース・マネージャは、すべてのデータベース更新を恒久的なものにします。同様に、トランザクションをロールバックすると、XA リソース・マネージャは、データベースをトランザクション開始前の状態に復元します。

XA リソース・マネージャの特徴は、ロールバック時のオブジェクト状態データの処理に関する設計の問題をより簡単なものにすることです。トランザクションに關与するオブジェクトは、コミットおよびロールバック権限を XA リソース・マネージャに委譲します。これによって、サーバ・アプリケーションのインプリメントは、大幅に単純化されます。

Transactions サンプル・アプリケーションのコンフィギュレーションの要件

University サンプル・アプリケーションは、Oracle のトランザクション・マネージャ・サーバ (TMS) を使用します。Oracle のデータベースを使用するには、サーバ・アプリケーションを構築する際に、Oracle から提供された特定のファイルを含める必要があります。

Transactions サンプル・アプリケーションの構築、設定、および実行の詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。また、オンライン・マニュアルにも、各サンプル・アプリケーション用の UBBCONFIG ファイルと、ファイルの各エントリの説明が示されています。

トランザクションの CORBA クライアントおよびサーバ・アプリケーションへの統合

BEA Tuxedo システムは、次のようにしてトランザクションをサポートします。

- クライアント・アプリケーションまたはサーバ・アプリケーションは、TransactionCurrent オブジェクトに対して呼び出しを実行することで、トランザクションを明示的に開始および終了できます。TransactionCurrent オブジェクトの詳細については、『BEA Tuxedo CORBA クライアント・アプリケーションの開発方法』および『BEA Tuxedo CORBA トランザクション』を参照してください。
- オブジェクトのインターフェイスにトランザクション方針を割り当てることができます。そうすることで、オブジェクトが呼び出されたときに、トランザクションが既に開始されていないならば、BEA Tuxedo システムがオブジェクトのために自動的にトランザクションを開始し、メソッド呼び出しが完了したときにトランザクションをコミットまたはロールバックすることができるようになります。すべてのトランザクションのコミット権限およびロールバック権限を XA リソース・マネージャに委譲する場合は、その XA リソース・マネージャおよびデータベースも考慮に入れて、オブジェクトのトランザクション方針を使用します。
- トランザクションに関与しているオブジェクトは、トランザクションを強制的にロールバックできます。つまり、トランザクションのスコープ内でオブジェクトが呼び出された後に、そのオブジェクトは

TransactionCurrent オブジェクト上で `rollback_only()` オペレーションを呼び出して、トランザクションをロールバック・オンリーとしてマークすることができます。これによって、現在のトランザクションがコミットされるのを防ぐことができます。エンティティ（通常はデータベース）が破損データまたは不正確なデータで更新される危険がある場合、オブジェクトは、トランザクションを `rollback` としてマークしなければならない場合があります。

- トランザクションに關与するオブジェクトは、最初に呼び出されたときから、トランザクションをコミットまたはロールバックする準備ができたときまでメモリに保持できます。間もなくコミットされるトランザクションの場合、これらのオブジェクトは、リソース・マネージャがこのトランザクションのコミットを準備する直前に BEA Tuxedo システムによってポーリングされます。ここでは、ポーリングとはオブジェクトの `Tobj_ServantBase::deactive_object()` オペレーションを呼び出して `reason` 値を渡すことを意味します。

オブジェクトがポーリングされる時、そのオブジェクトが TransactionCurrent オブジェクトの `rollback_only()` オペレーションを呼び出して現在のトランザクションのコミットを拒否する可能性があります。さらに、現在のトランザクションがロールバックされる場合、オブジェクトは、データベースへの書き込みをスキップできます。現在のトランザクションのコミットを拒否するオブジェクトがなければ、トランザクションはコミットされます。

以後の節では、必要なトランザクションの振る舞いをオブジェクトに指定するためにオブジェクトの活性化方針とトランザクション方針を使用する方法を説明します。これらの方針は、インターフェイスに適用されます。したがって、そのインターフェイスをインプリメントしているすべてのオペレーションとオブジェクトに適用されます。

注記 サーバ・アプリケーションが、トランザクションに参加するオブジェクトを管理している場合、そのアプリケーションの Server オブジェクトは、`TP::open_xa_rm()` オペレーションおよび `TP::close_xa_rm()` オペレーションを呼び出す必要があります。データベース接続の詳細については、第 6 章の 18 ページ「XA リソース・マネージャのオープン」を参照してください。

オブジェクトを自動的にトランザクションに関与させる方法

BEA Tuxedo システムは、`always` トランザクション方針を提供します。これは、オブジェクトが呼び出されたときにトランザクションがまだスコープ指定されていない場合、BEA Tuxedo システムがトランザクションを自動的に開始するように、そのオブジェクトのインターフェイスを定義します。そのオブジェクトの呼び出しが完了すると、BEA Tuxedo システムは、自動的にトランザクションをコミットまたはロールバックします。サーバ・アプリケーションもオブジェクト・インプリメンテーションも、この状態で `TransactionCurrent` オブジェクトを呼び出す必要はありません。つまり、BEA Tuxedo システムは、サーバ・アプリケーションの代わりに自動的に `TransactionCurrent` オブジェクトを呼び出します。

オブジェクトのインターフェイスに `always` トランザクション方針を割り当てるのが適切なのは、次の場合です。

- オブジェクトがデータベースに書き込みをし、このオブジェクトが呼び出されたときに、すべてのデータベースのコミットまたはロールバックの権限を XA リソース・マネージャに委譲する必要がある場合。
- クライアント・アプリケーションが、複数のオブジェクトに対して呼び出しを実行する大規模なトランザクションでオブジェクトを含めることができるようにし、呼び出しがすべて成功するか、呼び出しが失敗した場合にロールバックする必要がある場合。

オブジェクトを自動的にトランザクションに関与させる必要がある場合は、当該のオブジェクトのインターフェイスに関する次の方針をインプリメンテーション・コンフィギュレーション・ファイル (ICF ファイル) に記述します。

活性化方針	トランザクション方針
<code>process</code> 、 <code>method</code> 、または <code>transaction</code>	<code>always</code>

注記 データベース・カーソルは、複数のトランザクションにまたがることができません。CORBA の University サンプル・アプリケーションにある `CourseSynopsisEnumerator` オブジェクトでは、一致するコース概要を University データベースで検索するためにデータベース・カーソルが使用されています。データベース・カーソルは複数のトランザクションにまたがることができないので、`CourseSynopsisEnumerator` オブジェクトの `activate_object()` オペレーションは一致するすべてのコース概要をメモリに読み取ります。カーソルはイテレータ・クラスによって管理されているので、`CourseSynopsisEnumerator` オブジェクトからは見えないことに注意してください。

オブジェクトのトランザクションへの参加の有効化

オブジェクトをトランザクションのスコープ内で呼び出すことができるようにする必要がある場合、`optional` トランザクション方針をそのオブジェクトのインターフェイスに割り当てることができます。`optional` トランザクション方針は、データベース書き込みオペレーションを実行しないものの、トランザクション時に呼び出すことができるようにする必要があるオブジェクトに適しています。

オブジェクトに `optional` トランザクション方針を適用するために、そのオブジェクトのインターフェイス用の ICF ファイルで次の方針を指定できます。

活性化方針	トランザクション方針
<code>process</code> 、 <code>method</code> 、または <code>transaction</code>	<code>optional</code>

オブジェクトがデータベース書き込みオペレーションを実行しており、オブジェクトがトランザクションに関与できるようにする必要がある場合は、`always` トランザクション方針を割り当ての方が適切です。ただし、目的に応じて `optional` 方針を使用し、`TransactionCurrent` オブジェクトに対する呼び出しで書き込みオペレーションをカプセル化できます。つまり、オブジェクトがまだトランザクション内にスコープ指定されていない場合、データを書き込むオペレーション内で、トランザクションを開始およびコミットまた

はロールバックするために TransactionCurrent オブジェクトを呼び出し、write 文の周囲にトランザクションをスコープします。これによって、データベース書き込みオペレーションがトランザクションに関与する形で処理されます。また、性能も向上します。TransactionCurrent オブジェクトがトランザクションのスコープ内で呼び出されなければ、すべてのデータベース読み取りオペレーションがトランザクションの外部になるので、効率が高くなります。

注記 BEA Tuxedo システムで使用される XA リソース・マネージャの一部では、トランザクションに関与するすべてのオブジェクトについて、データベース書き込みオペレーションに加えて読み取りオペレーションについても、トランザクション内でスコープ指定する必要があります(しかし、それでも独自にトランザクションのスコープ指定はできます)。たとえば、BEA Tuxedo システムで Oracle TMS を使用する場合に、この要件が該当します。トランザクション方針を選択してオブジェクトに割り当てている場合、使用している XA リソース・マネージャの要件を把握します。

トランザクションのスコープ指定時のオブジェクト呼び出しの防止

多くの場合、オブジェクトをトランザクションから除外することは危険です。このようなオブジェクトがトランザクション時に呼び出されると、オブジェクトは例外を返し、トランザクションがロールバックされることがあります。BEA Tuxedo システムには `never` トランザクション方針が用意されていて、これをオブジェクトのインターフェイスに割り当てれば、現在のトランザクションが一時停止中でも、特定のオブジェクトをトランザクションの処理中に呼び出されないようにできます。

このトランザクション方針は、ロールバックできないディスクに永続状態を書き込むオブジェクトに適しています。たとえば、XA リソース・マネージャに管理されていないディスクにデータを書き込むオブジェクトなどです。クライアント・アプリケーションで、呼び出しの一部がトランザクションのスコープ指定を引き起こしているかどうかを認識できない場合、クライアント/サーバ・アプリケーションでこの機能を使用することは重要です。

したがって、トランザクションがスコープ指定されている場合、この方針を持つオブジェクトが呼び出されると、トランザクションをロールバックできるようにになります。

トランザクションのスコープ指定がされているときにオブジェクトの呼び出しを禁止するには、ICF ファイルで当該のオブジェクトのインターフェイスに次の方針を割り当てます。

活性化方針	トランザクション方針
process または method	never

実行中のトランザクションからのオブジェクトの除外

トランザクションの過程でオブジェクトの呼び出しを許可し、ただしそのオブジェクトをトランザクションの一部にはしないことがふさわしい場合もあります。このようなオブジェクトがトランザクションの最中に呼び出された場合、トランザクションは自動的に中断します。オブジェクトに対する呼び出しが完了すると、トランザクションは自動的に再開します。この目的のために、BEA Tuxedo システムには `ignore` トランザクション方針が用意されています。

`ignore` トランザクション方針は、通常はデータをディスクに書き込まないファクトリなどのオブジェクトに適している場合があります。ファクトリをトランザクションから除外することで、そのファクトリは、トランザクションの最中でもほかのクライアントの呼び出しに使用できるようになります。さらに、この方針を使用すると、トランザクションに関与しているオブジェクトを呼び出す際のオーバーヘッドが軽減されるので、サーバ・アプリケーションの処理効率が向上します。

トランザクションがオブジェクトに伝達されることを禁止するには、ICF ファイルで当該のオブジェクトのインターフェイスに次の方針を割り当てます。

活性化方針

トランザクション方針

process または method ignore

方針の割り当て

ICF ファイルを作成してオブジェクトに方針を指定する方法の詳細については、「第 2 章の 18 ページ「ステップ 4: オブジェクトのメモリ内での振り舞い」」を参照してください。

XA リソース・マネージャのオープン

オブジェクトのインターフェイスに `always` または `optional` トランザクション方針が適用されている場合、Server オブジェクトの

`Server::initialize()` オペレーションの `TP::open_xa_rm()` オペレーションを呼び出す必要があります。リソース・マネージャは、UBBCONFIG ファイルの `GROUPS` セクションにある `OPENINFO` パラメータで提供された情報を基にオープンされます。デフォルト・バージョンの `Server::initialize()` オペレーションは、自動的にリソース・マネージャをオープンします。

データをディスクに書き込まず、トランザクションに参加しているオブジェクト（通常、トランザクション方針は `optional`）がある場合、

`TP::open_xa_rm()` オペレーションへの呼び出しを含める必要があります。その呼び出しでは、ヌル・リソース・マネージャを指定します。

XA リソース・マネージャのクローズ

Server オブジェクトの `Server::initialize()` オペレーションが、XA リソース・マネージャをオープンする場合は、`Server::release()` オペレーションに以下の呼び出しを含めます。

```
TP::close_xa_rm();
```

トランザクション管理とオブジェクト状態管理

CORBA クライアントおよびサーバ・アプリケーションにトランザクションが必要な場合に、トランザクションとオブジェクト状態の管理を統合する方法は複数あります。一般に、BEA Tuxedo システムでは、オペレーション呼び出しの間のトランザクションについてスコープ指定を自動的に行うようにでき、このときにアプリケーションのロジックを変更したり、オブジェクトが永続状態をディスクに書き込む方法を変更したりする必要はありません。

ここでは、トランザクションとオブジェクト状態の管理に関する重要な項目の一部を説明します。

XA リソース・マネージャへのオブジェクト状態管理の委譲

XA リソース・マネージャを使用すると、たとえば CORBA University サンプル・アプリケーションで使用される Oracle のリソース・マネージャなどであれば、一般に、ロールバックでのオブジェクト状態データの処理に関連する設計上の問題が簡単になります。トランザクション・オブジェクトはコミットおよびロールバックの処理をいつでも XA リソース・マネージャに委譲できるので、このことによってサーバ・アプリケーションをインプリメントする作業が大幅に簡略化されます。つまり、トランザクションに關与するプロセス・バウンドまたはメソッド・バウンド・オブジェクトは、トランザクション時にデータベースに書き込みを実行し、トランザクションのロールバック時にリソース・マネージャに従ってデータベースに書き込まれたデータをロールバックできます。

トランザクションの作業が完了してから、データベースへの書き込みが始まるまでの待機

`transaction` 活性化方針は、トランザクションの作業が完了するまで書き込みたくない、または書き込めないメモリ内の状態をディスクに保持するオブジェクトに適しています。`transaction` 活性化方針をオブジェクトに割り当てると、オブジェクトは、以下ようになります。

- トランザクションの範囲内で最初に呼び出されたときにメモリに格納されます。
- トランザクションがコミットまたはロールバックされるまでメモリにとどまります。

トランザクションの作業が完了したら、BEA Tuxedo システムは各トランザクション・バウンド・オブジェクトの

`Tobj_ServantBase::deactivate_object()` オペレーションを呼び出して、`DR_TRANS_COMMITTING` または `DR_TRANS_ABORT` のどちらかの `reason` コードを渡します。変数が `DR_TRANS_COMMITTING` の場合、オブジェクトは、データベース書き込みオペレーションを呼び出すことができます。変数が `DR_TRANS_ABORT` であれば、オブジェクトは自身のデータベース書き込みオペレーションを省略します。

`transaction` 活性化方針のオブジェクトへの割り当ては、以下のような場合に適しています。

- トランザクションの作業が完了した時点で、オブジェクトの永続状態をディスクに書き込む必要がある。
ロールバックの対象となる可能性のあるデータベース書き込みオペレーションの数を減らすことができるので、これによって、パフォーマンスがより効率的になります。
- オブジェクトが、間もなくコミットされるトランザクションを拒否できるようにする場合。

BEA Tuxedo システムが `reason` 値として `DR_TRANS_COMMITTING` を渡す場合、オブジェクトは、必要であれば `TransactionCurrent` オブジェクトの `rollback_only()` オペレーションを呼び出すことができます。
`Tobj_ServantBase::deactivate_object()` オペレーションの内部で

`rollback_only()` オペレーションを呼び出した場合、その
`Tobj_ServantBase::deactivate_object()` オペレーションが再度呼び出
 される点に注意してください。

- 1つのトランザクションに複数回関与する可能性が高いオブジェクトが
 あり、そのトランザクション時にオブジェクトを連続して活性化または
 非活性化する際のオーバーヘッドを避けるようにする場合。

トランザクションのコミットを待機してからデータベースに書き込む機能を
 オブジェクトに付与するには、当該のオブジェクトのインターフェイスに関
 する次の方針を ICF ファイルに記述します。

活性化方針	トランザクション方針
transaction	always または optional

注記 トランザクション・バウンド・オブジェクトは、
`Tobj_ServantBase::deactivate_object()` オペレーション内でトラ
 ンザクションを開始したり、ほかのオブジェクトを呼び出したりす
 ることはできません。トランザクション・バウンド・オブジェクト
 が `Tobj_ServantBase::deactivate_object()` オペレーションの内部
 から実行できる唯一の有効な呼び出しは、データベースへの書き込
 みオペレーションのみです。

また、トランザクションに関与するオブジェクトがある場合、その
 オブジェクトを管理する Server オブジェクトは、管理されるオブ
 ジェクトがディスクに一切のデータを書き込まない場合でも、XA リ
 ソース・マネージャをオープンするための呼び出し、およびクロー
 ズするための呼び出しをそれぞれ含んでいる必要があります。(デー
 タをディスクに書き込まない、トランザクションに関与するオブ
 ジェクトがある場合、ヌル・リソース・マネージャを指定します)。
 XA リソース・マネージャをオープンする方法およびクローズする方
 法の詳細については、「第 6 章の 18 ページ「XA リソース・マネ
 ージャのオープン」」および「第 6 章の 18 ページ「XA リソース・マ
 ネージャのクローズ」」を参照してください。

BEA Tuxedo のトランザクションの使用に関する注意事項

CORBA クライアント / サーバ・アプリケーションにトランザクションを統合する際の注意事項は、以下のとおりです。

- 次のトランザクションは、BEA Tuxedo システムでは許可されません。
 - 入れ子になったトランザクション

既存のトランザクションが既に活性化している場合に、新しいトランザクションを開始することはできません。既存のトランザクションを中断すれば、新しいトランザクションを開始できます。しかし、中断したトランザクションを後で再開できるのは、トランザクションを中断したオブジェクトだけです。
 - 再帰的トランザクション

トランザクション・オブジェクトは、自身を呼び出すようなオブジェクトを呼び出せません。
- トランザクションを終了できるエンティティは、トランザクションを開始したオブジェクトのみです。厳密には、このオブジェクトはクライアント / アプリケーション、TP フレームワーク、またはオブジェクト (サーバ・アプリケーションによって管理されているもの) のいずれかです。トランザクションのスコープの内部で呼び出されたオブジェクトは、トランザクションを中断および再開できます。トランザクションが一時停止している間、オブジェクトはほかのトランザクションを開始および終了できます。しかし、トランザクションを開始しない限り、オブジェクト内でトランザクションを終了できません。
- オブジェクトは、一度に 1 つのトランザクションにのみ関与できます。BEA Tuxedo システムは並列トランザクションをサポートしていません。
- BEA Tuxedo システムは、現在トランザクションに関与しているオブジェクトへの要求をキューに入れません。トランザクションに関与しないクライアント・アプリケーションが、現在トランザクション内にあるオブジェクトに対してオペレーションを呼び出そうとすると、以下のエラー・メッセージが発行されます。

CORBA::OBJ_ADAPTER

トランザクション内にあるクライアントが、現在別のトランザクション内にあるオブジェクトに対してオペレーションを呼び出そうとすると、以下のエラー・メッセージが発行されます。

CORBA::INVALID_TRANSACTION

- トランザクション・バウンド・オブジェクトについては、すべての状態の処理を `Tobj_ServantBase::deactive_object()` オペレーションで行うことを検討してください。これにより、`Tobj_ServantBase::deactivate_object()` オペレーションが呼び出された時点でトランザクションの結果が分かるので、オブジェクトが自身の状態を適切に扱うことが簡単になります。

- 複数のオペレーションを持つものの、そのうちの少数のみがオブジェクトの永続状態に影響するメソッド・バウンド・オブジェクトの場合は、次の項目を検討する必要があります。

- トランザクション方針として `optional` を割り当てる。
- `TransactionCurrent` オブジェクトに対して呼び出しを実行することで、書き込みオペレーションをトランザクション内にスコープ指定する。

トランザクション外でオブジェクトが呼び出された場合、データの読み取りに対するトランザクションのスコープにオーバーヘッドは生じません。この方法で、トランザクション内でオブジェクトが呼び出されるかどうかに関係なく、オブジェクトのすべての書き込みオペレーションは、トランザクションに關与する形で処理されます。

- トランザクションのロールバックは非同期的です。したがって、トランザクション・コンテキストがアクティブな場合でも、オブジェクトを呼び出すことは可能です。このようなオブジェクトを呼び出そうとすると、例外が発生します。
- トランザクション方針として `always` を割り当てたオブジェクトが、クライアント・アプリケーションではなく BEA Tuxedo システムで開始されたトランザクションに關与している場合は、以下に注意してください。

オブジェクトのオペレーションの内部で例外が発生した場合、クライアント・アプリケーションは `OBJ_ADAPTER` 例外を受け取ります。この状況で、BEA Tuxedo システムは自動的にトランザクションをロールバックします。ただし、クライアント・アプリケーションは、トランザクシ

ンが BEA Tuxedo ドメインでスコープ指定されていないことをまったく認識していません。

- クライアント・アプリケーションがトランザクションを開始し、サーバ・アプリケーションがトランザクションにロールバックとマークして CORBA 例外を返した場合、クライアント・アプリケーションはトランザクション・ロールバックの例外のみを受け取り、CORBA 例外は受け取りません。

注記 WebLogic Enterprise バージョン 4.2 ソフトウェアでは、この状況を回避する方法がありません。トランザクションを開始する前に、アプリケーションができるだけ周到にデータの検証を実行するようにしてください。

- `TP::deactivateEnable` メソッドを持つトランザクション・オブジェクトに関する次の制限に注意してください。

トランザクション中に `TP::deactivateEnable` メソッドが呼び出された場合、オブジェクトはトランザクションの終了時に非活性化されます。しかし、`TP::deactivateEnable` メソッドが呼び出された時点とトランザクションがコミットされた時点の間に、オブジェクトで何らかのメソッドが呼び出された場合、そのオブジェクトは決して非活性化されません。

ユーザ定義の例外

Transactions サンプル・アプリケーションには、ユーザ定義例外 `TooManyCredits` のインスタンスが含まれています。クライアント・アプリケーションが学生をコースに登録しようとしたときに、学生が登録可能なコースの最大数を超過している場合、サーバ・アプリケーションはこの例外を返します。クライアント・アプリケーションは、この例外を受け取ると、学生をコースに登録するトランザクションをロールバックします。ここでは、CORBA クライアント / サーバ・アプリケーションでユーザ定義例外を定義およびインプリメントする方法について、`TooManyCredits` を例にして説明します。

CORBA クライアント / サーバ・アプリケーションにユーザ定義例外を含める作業には、次の手順が関与します。

1. OMG IDL ファイルで、例外を定義し、それを使用できるオペレーションを指定します。
2. インプリメンテーション・ファイルに例外をスローするコードを含めません。
3. クライアント・アプリケーションのソース・ファイルに、例外をキャッチして処理するコードを含めます。

次の節では、最初の 2 つの手順の説明と例を示します。

例外の定義

クライアント / サーバ・アプリケーションの OMG IDL ファイルでは、以下の作業を行います。

1. 例外の定義、および例外によって送信されるデータの定義。たとえば、`TooManyCredits` 例外は、学生が登録できる単位の最大数を表す `short` 型の整数値を渡すために定義します。したがって、`TooManyCredits` 例外の定義には、以下の OMG IDL 文が含まれます。

```
exception TooManyCredits
{
    unsigned short maximum_credits;
};
```

2. 例外をスローするオペレーションの定義で、例外を含めます。次の例は、`Registrar` インターフェイスの `register_for_courses()` オペレーションに対する OMG IDL 文を示したものです。

```
NotRegisteredList register_for_courses(
    in StudentId      student,
    in CourseNumberList courses)
    raises (TooManyCredits);
```

例外のスロー

例外を使用するオペレーションのインプリメンテーションでは、次の例のように、例外をスローするコードを記述します。

6 トランザクションの CORBA サーバ・アプリケーションへの統合

```
if ( ... ) {  
    UniversityZ::TooManyCredits e;  
    e.maximum_credits = 18;  
    throw e;  
}
```

7 BEA Tuxedo サービスの CORBA オブジェクトへのラッピング

この章では、Wrapper サンプル・アプリケーションを例にして、CORBA サーバ・アプリケーションによって管理されているオブジェクトの内部から BEA Tuxedo サービスを呼び出す方法の 1 つを概説します。

ここでは、以下の内容について説明します。

- BEA Tuxedo サービスのラッピングの概要
 - この節では、次の内容について説明します。
 - BEA Tuxedo サービスをラッピングするオブジェクトの設計
 - BEA Tuxedo サービス呼び出しをカプセル化するバッファの作成
 - BEA Tuxedo サービスとの間でメッセージを送信するオペレーションのインプリメンテーション
- Wrapper サンプル・アプリケーションの設計上の考慮事項

Wrapper サンプル・アプリケーションは、一連の課金用のオペレーションを BEA Tuxedo ATMI Teller アプリケーションに委譲します。このアプリケーションは、基本的な課金の手続きを実行する一連のサービスを含んでいます。この章で紹介するアプローチは、BEA Tuxedo アプリケーションを BEA Tuxedo ドメインに組み込むテクニックの一例です。

この章で示されている例では、CORBA オブジェクトのオペレーションと、アプリケーション内の特定のサービスへの呼び出しとの 1 対 1 の対応が示されます。つまり、BEA Tuxedo サービスへの呼び出しが CORBA オブジェクトのオペレーションとしてラッピングされるとも言えます。これは、オブジェクトが処理機能を BEA Tuxedo アプリケーションに委譲することです。BEA Tuxedo サービスのセットを CORBA サーバ・アプリケーションで使用する必要があるときには、この章で説明されるテクニックを試みてください。

この章では、BEA Tuxedo ATMI アプリケーションの詳細は説明しません。BEA Tuxedo ATMI アプリケーションをビルドおよびコンフィギュレーションする方法、および動作の詳細については、BEA Tuxedo のオンライン・マニュアルに含まれている BEA Tuxedo ATIMI の情報を参照してください。

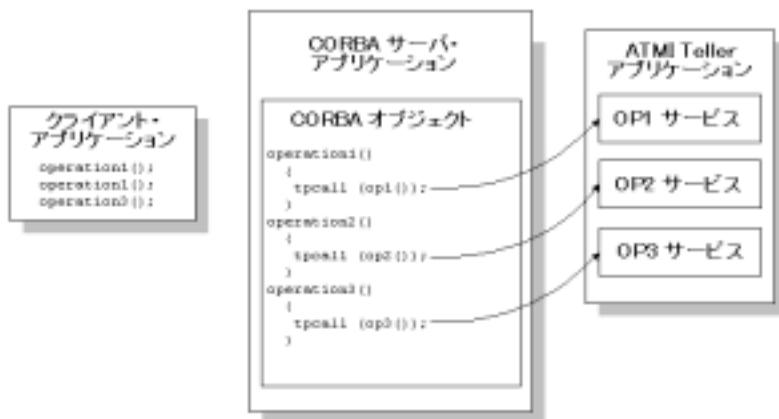
BEA Tuxedo サービスのラッピングの概要

この章で説明される、BEA Tuxedo サービスのセットをラッピングする処理には以下のステップがあります。

1. BEA Tuxedo システム向けの一連のタスクをオブジェクトのオペレーションとして構造化するオブジェクトを設計します。
2. BEA Tuxedo サービスによって使用されるメッセージ・バッファを作成します。このメッセージ・バッファは、BEA Tuxedo サービスとの間でメッセージの送信および受信に使用されます。このバッファは、アプリケーションのインプリメンテーション・ファイルでオブジェクトのコンストラクタに割り当てることができます。

- オブジェクト上に、BEA Tuxedo サービスとの間でメッセージを送信および受信するオペレーションをインプリメントします。このステップには、BEA Tuxedo サービスを呼び出す方法のインプリメンテーションを選択することも含まれます。

次の図は、クライアント・アプリケーション、CORBA サーバ・アプリケーションに管理されている CORBA オブジェクト、および BEA Tuxedo ATMI アプリケーションの間の関係を簡単に示します。BEA Tuxedo ATMI アプリケーションは、CORBA オブジェクトから呼び出されるサービスをインプリメントします。



BEA Tuxedo サービスをラッピングするオブジェクトの設計

この章で説明する第一のステップは、BEA Tuxedo ATMI アプリケーションへの呼び出しをラッピングするオブジェクトの設計です。たとえば、Wrapper サンプル・アプリケーションの目的は、学生の登録プロセスに課金の機能を追加することで、これは既存の BEA Tuxedo ATMI Teller アプリケーションに課金用オペレーションのセットを委譲すれば実現できます。

Wrapper サンプル・アプリケーションで使用されている BEA Tuxedo ATMI Teller アプリケーションには、次のサービスが含まれています。

- CURRBALANCE 指定された口座の現在の残高を取得します。
- CREDIT 指定された金額 (ドル) を口座に振り込みます。
- DEBIT 指定された金額 (ドル) を口座から引き落としします。

これらのサービスをラッピングするために、Wrapper サンプル・アプリケーションには新しいインターフェイス `Teller` を定義する別個の OMG IDL ファイルが含まれています。このインターフェイスには、次のオペレーションがあります。

- `get_balance()`
- `credit()`
- `debit()`

`Teller` オブジェクトのこれらの各オペレーションは、BEA Tuxedo ATMI `Teller` アプリケーションにあるサービスへの呼び出しと 1 対 1 でマップされています。

`Teller` オブジェクトの一般的な使用のシナリオは、以下のようになります。

1. クライアント・アプリケーションによって `Registrar` オブジェクトの `register_for_courses()` オペレーションが呼び出されて、このときに学生 ID が要求されます。
2. 登録の処理の一部として、`Registrar` オブジェクトによって `Teller` オブジェクトの `get_balance()` オペレーションが呼び出されて、口座番号が渡されます。
3. `Teller` オブジェクトの `get_balance()` オペレーションによって、口座番号がメッセージ・バッファに格納され、このバッファが `CURRBALANCE` サービスの BEA Tuxedo ATMI `Teller` アプリケーションに送信されます。
4. BEA Tuxedo ATMI `Teller` アプリケーションによって、メッセージ・バッファが受信され、その内容が取り出されてから、`CURRBALANCE` サービスに対して適切な呼び出しが行われます。
5. `CURRBALANCE` サービスによって、口座の現在の残高が `University` データベースから取得されて、BEA Tuxedo ATMI `Teller` アプリケーションに渡されます。

6. BEA Tuxedo ATMI Teller アプリケーションによって、現在の残高がメッセージ・バッファに挿入され、Teller オブジェクトに返されます。
7. Teller オブジェクトによって、現在の残高の合計がメッセージ・バッファから取り出された上で、現在の残高が Registrar オブジェクトに返されます。

Teller オブジェクトおよび Wrapper サンプル・アプリケーションの設計の詳細については、「第 7 章の 9 ページ「Wrapper サンプル・アプリケーションの設計上の考慮事項」」を参照してください。

BEA Tuxedo サービス呼び出しをカプセル化するバッファの作成

この章で説明される次のステップは、オブジェクトと BEA Tuxedo サービスの間でメッセージの送信に使用されるバッファの作成です。さまざまな BEA Tuxedo ATMI アプリケーションで使用可能なバッファ型は複数あり、この章の例では FML バッファ型に基づいたバッファを使用します。BEA Tuxedo システムのバッファ型の詳細については、BEA Tuxedo の情報を参照してください。

使用するアプリケーション・インプリメンテーション・ファイルで、選択したバッファ型を割り当てる必要があります。割り当てるバッファは Teller オブジェクトの特定のインスタンスについて一意である必要がないので、オブジェクトのコンストラクタに割り当てることができます。この割り当てのオペレーションには、一般に、バッファ型の指定、BEA Tuxedo サービスへのプロシージャ・コールに適した任意のフラグの受け渡し、およびバッファ・サイズの指定などが含まれます。

また、使用するインプリメンテーションのヘッダ・ファイルには、バッファを表す変数の定義を追加する必要もあります。

次のコード例では Wrapper アプリケーションの Teller オブジェクトのコンストラクタに、BEA Tuxedo のバッファ `m_tuxbuf` が割り当てられています。

```
Teller_i::Teller_i() :  
    m_tuxbuf((FBFR32*)tpalloc("FML32", "", 1000))  
{
```

```
    if (m_tuxbuf == 0) {  
        throw CORBA::INTERNAL();  
    }  
}
```

FML バッファを割り当てる行については、次のことに注意してください。

コード	説明
<code>tpalloc</code>	バッファを割り当てます。
<code>"FML32"</code>	FML バッファ型を指定します。
<code>" "</code>	一般に、BEA Tuxedo サービスに渡されるフラグを記述する部分です。この例では、渡されるフラグはありません。
<code>1000</code>	バッファ・サイズをバイト単位で指定します。

また、オブジェクトのインプリメンテーション・ファイルは、Wrapper アプリケーションのインプリメンテーション・ファイルにある次の文のようにして、デストラクタでバッファの割り当てを解除する必要があります。

```
tpfree((char*)m_tuxbuf);
```

BEA Tuxedo サービスとの間でメッセージを送信するオペレーションのインプリメンテーション

次のステップは、BEA Tuxedo ATMI アプリケーションへの呼び出しをラッピングするオブジェクトでのオペレーションをインプリメントすることです。このステップでは、オブジェクトから BEA Tuxedo サービスを呼び出す方法のインプリメンテーションを選択します。Wrapper サンプル・アプリケーションでは、`tpcall` インプリメンテーションが使用されます。

BEA Tuxedo サービスをラッピングするオブジェクトでのオペレーションには、次のことをする文が一般に含まれます。

- BEA Tuxedo サービスへ送信するデータでメッセージ・バッファを満たします。

- BEA Tuxedo サービスを呼び出します。呼び出しには、次の引数が含まれます。
 - a. 呼び出される BEA Tuxedo サービス
 - b. BEA Tuxedo サービスへ送信されるメッセージ・バッファ
 - c. BEA Tuxedo サービスから返されるメッセージ・バッファ
 - d. BEA Tuxedo サービスの応答が格納されるバッファのサイズ
- BEA Tuxedo サービスからの応答を取り出します。
- 結果をクライアント・アプリケーションに返します。

次の例は、Wrapper アプリケーションの Teller オブジェクトでの `get_balance()` オペレーションのインプリメンテーションを示します。このオペレーションでは特定の口座の残高が取得され、BEA Tuxedo サービス `CURRBALANCE` が呼び出されます。

```
CORBA::Double Teller_i::get_balance(BillingW::AccountNumber account)
{
    // 「in」パラメータ ( 口座番号 ) をマーシャリングする
    Fchg32(m_tuxbuf, ACCOUNT_NO, 0, (char*)&account, 0);
    long size = Fsizeof32(tuxbuf);
    // CURRBALANCE Tuxedo サービスを呼び出す
    if (tpcall("CURRBALANCE", (char*)tuxbuf, 0,
              (char**)&tuxbuf, &size, 0) ) {
        throw CORBA::PERSIST_STORE();
    }
    // 「out」パラメータ ( 現在の残高 ) のマーシャリングを解除する
    CORBA::Double currbal;
    Fget32(m_tuxbuf, CURR_BALANCE, 0, (char*)&currbal, 0);
    return currbal;
}
```

次のコード例にある文は、メッセージ・バッファ `m_tuxbuf` を学生の口座番号で満たします。FML の詳細については、BEA Tuxedo ATMI FML Function Reference を参照してください。

```
Fchg32(m_tuxbuf, ACCOUNT_NO, 0, (char*)&account, 0);
```

次の文は、`tpcall` のインプリメンテーションを通じて `CURRBALANCE` BEA Tuxedo サービスを呼び出し、メッセージ・バッファを渡します。また、この文は、BEA Tuxedo サービスの応答が格納される場所も指定します。この例では、要求の送信元となったバッファと同じバッファです。

```
if (tpcall("CURRBALANCE", (char*)tuxbuf, 0,
          (char**)&tuxbuf, &size, 0) ) {
    throw CORBA::PERSIST_STORE();
}
```

次の文は、返された BEA Tuxedo メッセージ・バッファから残高を取り出します。

```
Fget32(m_tuxbuf, CURR_BALANCE, 0, (char*)&currbal, 0);
```

`get_balance()` オペレーションの最後の行で、クライアント・アプリケーションへ結果が返されます。

```
return currbal;
```

制限事項

BEA Tuxedo ドメイン内に BEA Tuxedo サービスを組み込む方法については、次の制限事項に注意してください。

- オブジェクトのインプリメンテーションと BEA Tuxedo サービスを同一のサーバ・アプリケーション内で組み合わせることはできません。BEA Tuxedo サービスは、CORBA サーバ・アプリケーションと同一のドメインにある別個の BEA Tuxedo サーバ・アプリケーション内にのみ存在できます。
- BEA Tuxedo サービスを呼び出すオブジェクト内に `tpreturn()` または `tpforward()` の BEA Tuxedo インプリメンテーションを含めることはできません。

Wrapper サンプル・アプリケーションの設計上の考慮事項

Wrapper サンプル・アプリケーションの基本的な設計上の考慮事項は、この節で説明するシナリオに基づいています。学生がコースを登録するとき、`Registrar` オブジェクトが登録プロセスの一部として `Teller` オブジェクトの呼び出しを実行し、`Teller` オブジェクトは学生の口座にコースの課金をします。

ここでは、Wrapper サンプル・アプリケーションの設計について説明し、`Billing` というサーバ・アプリケーションをコンフィギュレーションに追加します。このため、Wrapper サンプル・アプリケーションには次の4つのサーバ・アプリケーションがあります。

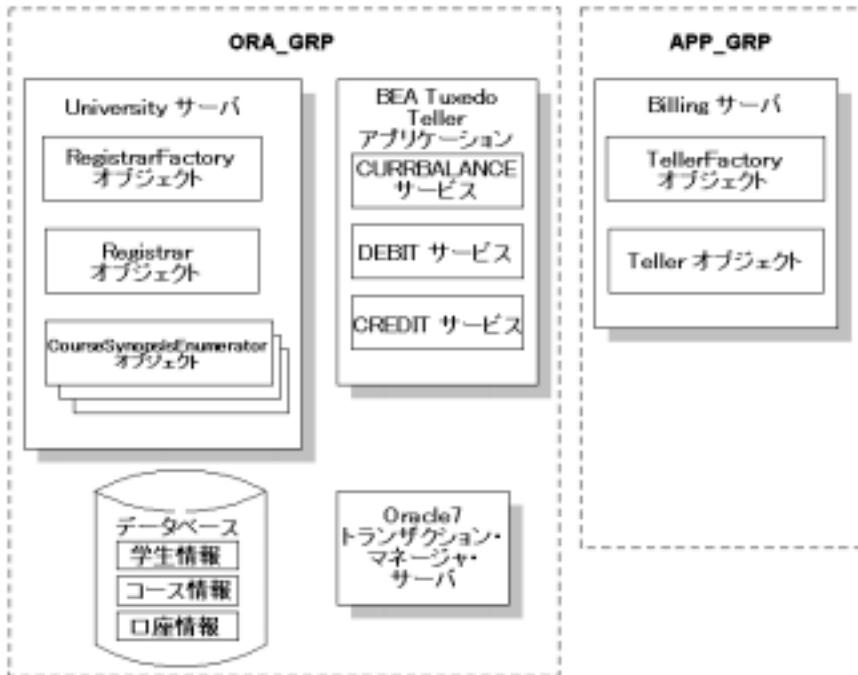
- `University`、`RegistrarFactory`、`Registrar`、および `CourseSynopsisEnumerator` オブジェクトがあります。
- `Billing`、`TellerFactory` および `Teller` オブジェクトがあります。
- `BEA Tuxedo ATMI Teller` アプリケーション。`CURRBALANCE`、`CREDIT`、および `DEBIT` サービスがあります。
- `Oracle` トランザクション・マネージャ・サーバ (TMS)。

さらに、Wrapper サンプル・アプリケーションの `UBBCONFIG` ファイルで、次のグループが指定されます。

- `ORA_GRP`。 `University` サーバ・アプリケーション、`BEA Tuxedo ATMI Teller` アプリケーション、および `Oracle TMS` があります。これら3つのプロセスは、`University` データベースでのトランザクションに関係するので、データベースと共に同じグループに存在する必要があります。
- `APP_GRP`。 `Billing` サーバ・アプリケーションがあります。`Billing` アプリケーションは `University` データベースとのやり取りをしないので、このアプリケーションが `ORA_GRP` に存在する必要はありません。

7 BEA Tuxedo サービスの CORBA オブジェクトへのラッピング

次の図に、Wrapper サンプル・アプリケーションでの BEA Tuxedo ドメインのコンフィギュレーションを示します。



BEA Tuxedo ATMI アプリケーションを University サンプル・アプリケーションに組み込むことは、Process-Entity デザイン・パターンを使用する観点からは合理的です。一般に BEA Tuxedo ATMI アプリケーションは Process-Entity デザイン・パターンをインプリメントしますが、このデザイン・パターンは University サンプル・アプリケーションでも使用されています。

University データベースが更新されて、各学生の口座情報が格納されている新しいテーブルが含まれます。このため、BEA Tuxedo ATMI Teller アプリケーションのサービスで課金データを処理する際には、University データベースを使用してトランザクションが実行されます。

Wrapper University サンプル・アプリケーションのしくみ

Wrapper サンプル・アプリケーションでの一般的な使用シナリオでは、以下のようなイベントのシーケンスを経ます。

1. 学生のログオン手続きの後で、クライアント・アプリケーションによって Registrar オブジェクトの `get_student_details()` オペレーションが呼び出されます。`get_student_details()` オペレーションのインプリメンテーションに含まれているコードによって、次の内容が取得されます。
 - 学生の口座番号。データベースにある学生のテーブルから。
 - 学生の残高。データベースの口座テーブルから。Teller オブジェクト上で `get_balance()` オペレーションを呼び出して取得されます。
2. 学生は、Basic サンプル・アプリケーションの場合と同様にコースを検索して、登録を希望するコースのリストを作成します。
3. クライアント・アプリケーションは、Transaction サンプル・アプリケーションのシナリオと同様に、Registrar オブジェクトに要求を送信して `register_for_courses()` オペレーションを呼び出します。この要求にコース番号と学生 ID のみが含まれる点も同じです。
4. 学生ごとにコースのリストを登録する際に、`register_for_courses()` オペレーションは次の処理を呼び出します。
 - Teller オブジェクトの `get_balance()` オペレーション。学生の口座に滞納がないことを確認します。
 - Teller オブジェクトの `debit()` オペレーション。Billing サーバ・アプリケーションによって管理され、コースの課金を行います。
5. Teller オブジェクトの `get_balance()` および `debit()` オペレーションのそれぞれが、BEA Tuxedo ATMI Teller アプリケーションへの要求を送信します。この要求には、適切な呼び出しが格納された FML バッファがカプセル化されています。これには、BEA Tuxedo ATMI Teller アプリケーションの CURRBALANCE および DEBIT サービスそれぞれへの口座番号呼び出しも含まれます。
6. CURRBALANCE および DEBIT サービスは、それぞれ、適切なデータベース呼び出しを実行して、現在の残高を取得し、学生が登録したコースの課金を反映するように学生の口座から引き落とします。

学生の口座に滞納があれば、Registrar オブジェクトはクライアント・アプリケーションに DelinquentAccount 例外を返します。その際、クライアント・アプリケーションはトランザクションをロールバックします。

debit() オペレーションが失敗した場合、Teller オブジェクトは TransactionCurrent オブジェクトの rollback_only() オペレーションを呼び出します。Teller および Registrar オブジェクトは同一のトランザクション内にスコープ指定されているため、このロールバックは、登録プロセスの全体に影響することで、データベースの不整合(たとえば、学生がコースに登録したのに、学生の口座残高からコースの分が引き落とされていないなど)を防ぎます。

7. 例外が発生しなければ、Registrar オブジェクトによって学生は希望のコースに登録されます。

Billing サーバ・アプリケーションのインターフェイス定義

以下のインターフェイス定義が、Billing サーバ・アプリケーション用に定義されています。

- TellerFactory オブジェクト。その唯一のオペレーションは find_teller() です。find_teller() オペレーションは、University サーバの RegistrarFactory オブジェクトの find_registrar() オペレーションとまったく同じ動作をします。
- Teller オブジェクト。前述のように、このオブジェクトは次のオペレーションをインプリメントします。
 - debit()
 - credit()
 - current_balance()

Registrar オブジェクトと同様に、Teller オブジェクトには状態データがなく、一意なオブジェクト ID (OID) もありません。

Wrapper サンプル・アプリケーションの設計上の追加考慮事項

次の追加考慮事項が、Wrapper サンプル・アプリケーションの設計に影響します。

- Registrar オブジェクトには、課金用オペレーションを扱う Teller オブジェクトへ要求を送信する方法が必要です。
- University サーバ・アプリケーションおよび BEA Tuxedo ATMI Teller アプリケーションは、同じデータベースにアクセスできる必要があります。このため、コース登録のトランザクションが適正に動作するには、2つのサーバ・アプリケーションの両方が Oracle TMS および University データベースと同じサーバ・グループに属している必要があります。

こうした考慮事項の両方により、Wrapper サンプル・アプリケーションの UBBCONFIG ファイルが重要になります。以後の節では、これに関係する設計上の追加考慮事項を詳細に説明します。

Teller オブジェクトへの要求の送信

ここまでは、University サーバ・アプリケーションのすべてのオブジェクトは同じサーバ・プロセス内で定義されていました。そのため、1つのオブジェクトがほかのオブジェクトに要求を送信する処理は明解でした。次のステップでは、Registrar オブジェクトおよび CourseSynopsisEnumerator オブジェクトを例にして示します。

1. Registrar オブジェクトが、CourseSynopsisEnumerator オブジェクトへのオブジェクト・リファレンスを作成します。
2. 新しく作成されたオブジェクト・リファレンスを使用して、Registrar オブジェクトは CourseSynopsisEnumerator オブジェクトへの要求を送信します。
3. CourseSynopsisEnumerator オブジェクトがメモリに存在しない場合は、TP フレームワークが Server オブジェクトの `Server::create_servant()` オペレーションを呼び出して、CourseSynopsisEnumerator オブジェクトをインスタンス化します。

しかし、2つのサーバ・プロセスが実行中で、1つのプロセスにあるオブジェクトが2つめのプロセスによって管理されているオブジェクトに要求を送信する必要がある場合、その手順はやや複雑です。たとえば、別のサーバ・プロセスにあるオブジェクトへのオブジェクト・リファレンスを取得するには、重要な前提があります。1つは、2番目のサーバ・プロセスの実行中に要求を行うことです。さらに、別のサーバ・プロセスにあるオブジェクト用のファクトリが利用可能である必要があります。

Wrapper サンプル・アプリケーションでは、これを解決するために以下のコンフィギュレーションおよび設計の要素を組み込んでいます。

- University サーバ・アプリケーションは、University Server オブジェクトの `Server::initialize()` オペレーションにある `TellerFactory` オブジェクトへのオブジェクト・リファレンスを取得します。次に、University サーバ・アプリケーションは `TellerFactory` へのオブジェクト・リファレンスをキャッシュします。これにより、`Registrar` オブジェクトが `TellerFactory` を必要とするたびに次の処理を実行せずに済むので、性能の最適化に貢献します。
 - `Bootstrap` オブジェクトの `resolve_initial_references()` オペレーションを呼び出して、`FactoryFinder` オブジェクトを取得します。
 - `FactoryFinder` オブジェクトの `find_one_factory_by_id()` オペレーションを呼び出して、`TellerFactory` オブジェクトへのリファレンスを取得します。
- `Billing` サーバ・プロセスが、University サーバ・プロセスよりも先に開始されます。続いて `Registrar` オブジェクトによって `TellerFactory` オブジェクトが呼び出されると、`Registrar` オブジェクトは `Server::initialize()` オペレーション (前項で説明したオペレーション) によって取得されたオブジェクト・リファレンスを使用します。`UBBCONFIG` ファイルは、各サーバ・プロセスの起動順に指定してください。
- コース登録のプロセス中に課金を処理するために、`Registrar` オブジェクトの `register_for_courses()` および `get_student_details()` オペレーションは、`Teller` オブジェクトのオペレーションを呼び出すコードを含むように変更されます。

例外の処理

Wrapper サンプル・アプリケーションは、学生による課金の総額が限度を超えた場合を処理できるように設計されています。学生が University で許される限度を超えてコースを登録しようとした場合、`Registrar` オブジェクトはユーザ定義の `DelinquentAccount` 例外を生成します。この例外がクライアント・アプリケーションに返されると、クライアント・アプリケーションに

よってトランザクションがロールバックされます。ユーザ定義例外をインプリメントする方法の詳細については、「第 6 章の 24 ページ「ユーザ定義の例外」」を参照してください。

Wrapper サンプル・アプリケーションのインターフェイスに対するトランザクション方針の設定

Wrapper サンプル・アプリケーションの性能に影響するもう 1 つの考慮事項は、アプリケーションのオブジェクトのインターフェイスに適したトランザクション方針を設定することです。Registrar、CourseSynopsisEnumerator、および Teller オブジェクトは、always トランザクション方針によってコンフィギュレーションされます。RegistrarFactory および TellerFactory オブジェクトは ignore トランザクション方針によってコンフィギュレーションされます。これらのオブジェクトには、トランザクションに含まれる必要がないので、トランザクションのコンテキストが伝達されないようになります。

University および Billing サーバ・アプリケーションのコンフィギュレーション

前述のように、Billing サーバ・アプリケーションは、University データベースおよび University アプリケーション、BEA Tuxedo ATMI Teller アプリケーション、および Oracle トランザクション・マネージャ・サーバ (TMS) アプリケーションを含むグループとは別のグループでコンフィギュレーションされます。

しかし、Billing サーバ・アプリケーションは学生をコースに登録するトランザクションに参加するので、Billing サーバ・アプリケーションは Server オブジェクトの `TP::open_xa_rm()` および `TP::close_xa_rm()` オペレーションへの呼び出しを含む必要があります。これは、任意のトランザクションに含まれるオブジェクトを管理するあらゆるサーバ・アプリケーションに対する要件です。そのようなオブジェクトがデータベースに対する読み取りまたは書き込みのオペレーションを一切実行しない場合は、次の場所に `ル・リソース・マネージャ` を指定することができます。

- `UBBCONFIG` ファイル内の適切なグループ定義
- サーバ・アプリケーションをビルドする際の `buidobjserver` コマンドの引数

7 BEA Tuxedo サービスの CORBA オブジェクトへのラッピング

Wrapper サンプル・アプリケーションの構築、設定、および実行の詳細については、『BEA Tuxedo CORBA University サンプル・アプリケーション』を参照してください。

8 BEA Tuxedo の CORBA サーバ・ア プリケーションのス ケーリング

この章では、BEA Tuxedo システムの主要なスケーラビリティ機能を利用して CORBA サーバ・アプリケーションのスケーラビリティを高める方法について、Production University サンプル・アプリケーションを例にして説明します。Production サンプル・アプリケーションは、これらのスケーラビリティ機能を利用して以下の目標を達成します。

- 並列処理機能を追加して、BEA Tuxedo ドメインで複数のクライアント要求を同時に処理できるようにします。
- Production サンプル・アプリケーションのサーバ・アプリケーション上の処理負荷を複数のマシンに分散させます。

ここでは、以下の内容について説明します。

- BEA Tuxedo システムで利用可能なスケーラビリティ機能の概要
- BEA Tuxedo サーバ・アプリケーションのスケーリング . この節では、次の内容について説明します。
 - サーバ・プロセスおよびサーバ・グループの複製
 - オブジェクト状態管理によるアプリケーションのスケーリング

- ファクトリ・ベース・ルーティング
- Production サーバ・アプリケーションをさらにスケールングする方法
- 状態を持たないオブジェクトまたは状態を持つオブジェクトの選択

BEA Tuxedo システムで利用可能なスケラビリティ機能の概要

高度にスケラブルなアプリケーションのサポートは、BEA Tuxedo システムの長所の 1 つです。多くのアプリケーションは、1 ~ 10 のサーバ・プロセス、および 10 ~ 100 のクライアント・アプリケーションが動作している環境で良好に機能します。しかし、ビジネス環境では、アプリケーションは次のレベルの処理をサポートする必要があります。

- 数百のサーバ・プロセス
- 数万のクライアント・アプリケーション
- 数百万のオブジェクト

このような要件を負ったアプリケーションをデプロイすると、リソースの不足および性能のボトルネックがすぐに表面化します。BEA Tuxedo システムではこうした大規模なデプロイメントが複数の方法でサポートされていて、この章ではそれらのうち次の 3 つについて説明します。

- サーバ・プロセスおよびサーバ・グループの複製
- オブジェクト状態管理
- ファクトリ・ベース・ルーティング

アプリケーションを高度にスケラブルにするために BEA Tuxedo システムで提供されているほかの機能には、IOP リスナ / ハンドラがあります。概要については『BEA Tuxedo CORBA アプリケーション入門』、詳細な説明につ

いては『BEA Tuxedo アプリケーションの設定』を参照してください。また、『BEA Tuxedo CORBA アプリケーションのスケールリング、分散、およびチューニング』も参照してください。

BEA Tuxedo サーバ・アプリケーションのスケールリング

ここでは、非常に大規模な処理機能を実現できるようにアプリケーションをスケールリングする方法について、Production サンプル・アプリケーションを例にして説明します。Production サンプル・アプリケーションの設計上の基本的な目標は、対応できるクライアント・アプリケーションの数を大幅に増やすために、次のように対処することです。

- 同じインターフェイスをインプリメントする複数のオブジェクト上のクライアント要求を 1 台のマシン上で並行処理します。
- 一部の学生の要求を 1 台のマシンに割り当て、残りの学生の要求は別のマシンに割り当てます。
- 処理負荷を分担するためのマシンを追加します。

こうした設計目標に対応するために、Production サンプル・アプリケーションでは次の処理をします。

- University、Billing、および BEA Tuxedo Teller アプリケーションのサーバ・プロセスを、それらがコンフィギュレーションされているグループ内で複製します。
- 上記のグループを追加のマシン上に複製します。
- サーバ・プロセスが同時に管理できるクライアント要求の数を増やすために、状態を持たないオブジェクト・モデルをインプリメントします。
- 次のオブジェクトに一意的なオブジェクト ID (OID) を割り当てて、それらのオブジェクトを各自のグループで同時に複数回インスタンス化できるようにします。これにより、次のオブジェクトはプロセス単位ではなく

クライアント・アプリケーション単位で利用可能になり、並行処理の機能に対応できます。

- RegistrarFactory
 - Registrar
 - TellerFactory
 - Teller
- 一部の学生の要求を 1 台のマシンに割り当て、別の学生の要求をほかのマシンに割り当てるために、ファクトリ・ベース・ルーティングをインプリメントします。

注記 Production サンプル・アプリケーションを使いやすくするために、このアプリケーションは 1 台のマシン上で 1 つのデータベースを使用して実行できるように BEA Tuxedo ソフトウェア・キット上でコンフィギュレーションされています。ただし、この章で示した例では、アプリケーションは 2 つのデータベースを使って 2 つのマシン上で動作しています。

Production サンプル・アプリケーションの設計では、複数台のマシン上で複数のデータベースを使用して実行するコンフィギュレーションが可能になるように設定されています。複数のマシンおよびデータベースを使用するようにコンフィギュレーションを変更する作業には、UBBCONFIG ファイルの変更とデータベースの分離が関係し、これらの詳細は第 8 章の 27 ページ「Production サーバ・アプリケーションをさらにスケールリングする方法」で説明されています。

以後の各節では、Production サンプル・アプリケーションのスケラビリティの目標を達成するために、複製されたサーバ・プロセスおよびサーバ・グループの使用、オブジェクト状態管理、およびファクトリ・ベース・ルーティングについて説明します。次の節では、Production サンプル・アプリケーションにインプリメントされている OMG IDL の変更について説明します。

Production サンプル・アプリケーションの OMG IDL の変更

Production サンプル・アプリケーションでの OMG IDL の変更は、RegistrarFactory オブジェクトの `find_registrar()` オペレーション、および TellerFactory オブジェクトの `find_teller()` オペレーションに限定されています。これら 2 つのオペレーションはそれぞれ、ファクトリ・ベース・ルーティングのインプリメンテーションに必要な学生 ID と口座番号を要求するように変更されます。Production サンプル・アプリケーションでのファクトリ・ベース・ルーティングのインプリメンテーションおよび使用については、第 8 章の 14 ページ「ファクトリ・ベース・ルーティング」を参照してください。

サーバ・プロセスおよびサーバ・グループの複製

BEA Tuxedo システムでは、サーバ・アプリケーションのコンフィギュレーションに多様な選択肢が用意されています。たとえば、以下のコンフィギュレーションが可能です。

- 1 台のマシンに、1 つのインターフェイスをインプリメントする 1 つのサーバ・プロセス。
- 1 台のマシンに、1 つのインターフェイスをインプリメントする複数のサーバ・プロセス。
- 1 台のマシンに、複数のインターフェイスをインプリメントする複数のサーバ・プロセス。ファクトリ・ベース・ルーティングの有無は任意。
- 複数台のマシンに複数のインターフェイスと複数のサーバ・プロセス。ファクトリ・ベース・ルーティングの有無は任意。

以上をまとめると、次のようになります。

- クライアント / サーバ・アプリケーションに並行処理の機能を追加するには、使用するサーバ・プロセスを複製します。

- デプロイメント環境にマシンを追加するには、グループを追加した上でファクトリ・ベース・ルーティングをインプリメントします。

次の節では、複製されたサーバ・プロセスおよびグループについて説明するほか、それらを BEA Tuxedo システムにコンフィギュレーションする方法についても説明します。

複製されたサーバ・プロセス

使用するアプリケーションでサーバ・プロセスを複製すると、以下の利点があります。

- サーバ・アプリケーションに着信する要求の負荷のバランシングを行う方法が得られます。BEA Tuxedo ドメインのサーバ・グループへの要求が着信すると、BEA Tuxedo システムはその要求をグループ内で負荷が最も低いと見なされるサーバヘルーティングします。
- サーバ・アプリケーションの性能が改善されます。一度に 1 つのクライアント要求を処理できる 1 つのサーバ・プロセスを持つ代わりに、複数のクライアント要求を同時に処理できる複数のサーバ・プロセスを利用可能にできます。この方法が正常に機能するには、各オブジェクトが一意である必要があります。そのためには、サーバ・アプリケーションのファクトリで一意的な OID を割り当ててください。
- サーバ・イメージの 1 つが停止した場合でも、有効なフェイルオーバー保護が得られます

複製されたサーバ・プロセスの利点を完全に活用するには、使用するサーバ・アプリケーションによってインスタンス化される各オブジェクトのすべてが一意的な ID を持つようにしてください。こうすれば、オブジェクト上でのクライアント呼び出しによって必要なオブジェクトがインスタンス化される処理が、利用可能な複数のサーバ・プロセスの境界内で実行されるので、既に活性化されているオブジェクトのためのキューに入れられずに済みません。

図 8-1 は、次のことを示します。

- University サーバ・アプリケーション、BEA Tuxedo Teller アプリケーション、および Oracle TMS の各サーバ・プロセスは、ORA_GRP グループ内で複製されます。

- Billing サーバ・プロセスは、APP_GRP グループ内で複製されます。

この図では、2つのグループは1台のマシンで実行されているものとして示されます。

図 8-1 Production サンプルの複製サーバ・グループ



これらのグループのいずれかに要求が着信したとき、BEA Tuxedo ドメインには要求を処理できる利用可能なサーバ・プロセスが複数あり、BEA Tuxedo ドメインは最も負荷の少ないサーバ・プロセスを選択することができます。

図 8-1 では、次の点に注意してください。

- 常に、特定のサーバ・プロセス内で、RegistrarFactory、Registrar、TellerFactory、または Teller オブジェクトのインスタンスが複数存在することはできません。

- `CourseSynopsisEnumerator` オブジェクトは、どの University サーバ・プロセスでも、任意の数だけ存在することができます。

複製されたサーバ・グループ

サーバ・グループの概念は BEA Tuxedo システムに固有のもので、CORBA のインプリメンテーションに追加されます。サーバ・グループは、BEA Tuxedo システムのスケラビリティ機能の重要な要素です。基本的に、デプロイメント・システムにマシンを追加する場合は、グループを追加する必要があります。

図 8-2 は、別のマシンに複製された Production サンプル・アプリケーションのグループを示します。このアプリケーションの `UBBCONFIG` ファイルに指定されているとおり、`ORA_GRP2` と `APP_GRP2` があります。

図 8-2 複数のマシンにわたるサーバ・グループの複製

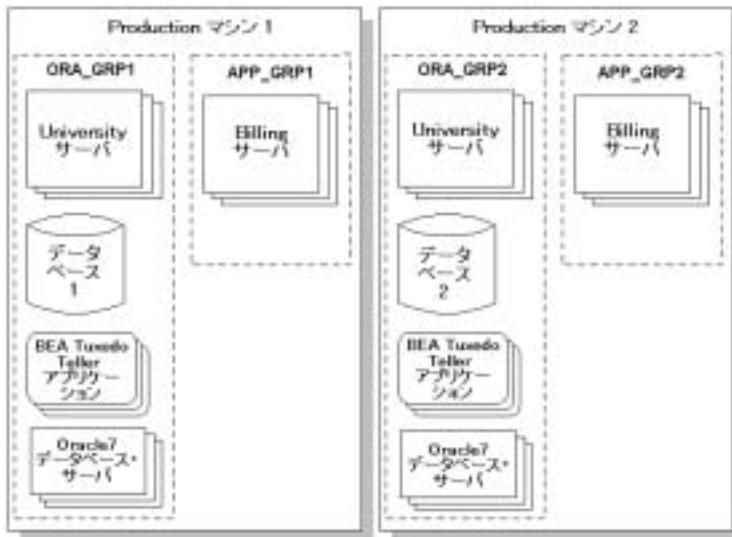


図 8-2 では、Production マシン 1 と 2 のグループの内容で異なるのはデータベースのみです。Production マシン 1 用のデータベースには、学生のサブセットについて、学生情報および口座情報が含まれています。Production マシン 2 用のデータベースには、学生の別のサブセットについて、学生情報お

よび口座情報が含まれています。両方のデータベースにあるコース情報のテーブルは同じ内容です。データベースにある学生情報が、同じデータベース内の口座情報とは完全に無関連であることに注意してください。

サーバ・グループのコンフィギュレーションの方法、実行場所、および複製の方法は、UBBCONFIG ファイルに指定されています。サーバ・グループを複製すると、次の処理が可能になります。

- 任意のアプリケーションまたはアプリケーションのセットに関する処理負荷を、別のマシンにも分散する方法があります。
- ファクトリ・ベース・ルーティングを使用して、任意のインターフェイスへの要求の 1 セットを 1 台のマシンに送信し、同じインターフェイスへの要求の別のセットは別のマシンへ送ることができます。

複数のサーバ・グループを持つと、次のような影響があります。

- クライアント要求が BEA Tuxedo ドメインに着信したとき、BEA Tuxedo システムはオブジェクト・リファレンスに指定されたグループ ID を確認します。
- BEA Tuxedo ドメインは、要求がルーティングされたグループ内で最も負荷の低い、要求を処理できるサーバ・プロセスに要求を送信します。

Production サンプル・アプリケーションがファクトリ・ベース・ルーティングを使用して複数台のマシンにアプリケーションの処理負荷を分散させる方法の詳細については、第 8 章の 14 ページ「ファクトリ・ベース・ルーティング」を参照してください。

複製されたサーバ・プロセスおよびグループのコンフィギュレーション

使用する BEA Tuxedo ドメインにある複製されたサーバ・プロセスおよびグループをコンフィギュレーションするには、以下の手順に従います。

1. アプリケーションの UBBCONFIG ファイルを、「ワードパッド」などのテキスト・エディタで開きます。
2. GROUPS セクションで、コンフィギュレーションするグループの名前を指定します。

3. `SERVERS` セクションに、複製するサーバ・プロセスに関する次の情報を入力します。
 - サーバ・アプリケーションの名前。
 - `GROUP` パラメータ。サーバ・プロセスが属するグループの名前を指定します。複数のグループに関係するサーバ・プロセスを複製する場合は、各グループに1つずつサーバ・プロセスを指定します。
 - `SRVID` パラメータ。数値識別子を指定して、サーバ・プロセスに一意な ID を割り当てます。
 - `MIN` パラメータ。アプリケーションの起動時に開始されるサーバ・プロセスのインスタンスの数を指定します。
 - `MAX` パラメータ。同時に実行できるサーバ・プロセスの最大数を指定します。

`MIN` および `MAX` パラメータは、指定されたオブジェクトへの要求をサーバ・アプリケーションでどの程度まで並行処理できるかを決定します。実行時には、必要に応じて、システム管理者がリソースのボトルネックを調べて、新しいサーバ・プロセスを起動することができます。このアプリケーションは、システム管理者がスケールリングできるように設計されています。

次に、Production サンプル・アプリケーションの `UBBCONFIG` ファイルの `GROUPS` および `SERVERS` セクションから引用した行を例示します。

```
*GROUPS
APP_GRP1
  LMID      = SITE1
  GRPNO     = 2
  TMSNAME   = TMS
APP_GRP2
  LMID      = SITE1
  GRPNO     = 3
  TMSNAME   = TMS
ORA_GRP1
  LMID      = SITE1
  GRPNO     = 4
  OPENINFO  = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
  CLOSEINFO = " "
  TMSNAME   = "TMS_ORA"
ORA_GRP2
  LMID      = SITE1
  GRPNO     = 5
```

```
OPENINFO = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
CLOSEINFO = ""
TMSNAME = "TMS_ORA"

*SERVERS
# デフォルトでは、各サーバのインスタンスを 2 つ活性化
# して、管理者が各サーバのインスタンスを 5 つ
# まで活性化できるようにしている
DEFAULT:
    MIN = 2
    MAX = 5
tello_server
    SRVGRP = ORA_GRP1
    SRVID = 10
    RESTART = N
tello_server
    SRVGRP = ORA_GRP2
    SRVID = 10
    RESTART = N
billp_server
    SRVGRP = APP_GRP1
    SRVID = 10
    RESTART = N
billp_server
    SRVGRP = APP_GRP2
    SRVID = 10
    RESTART = N
univp_server
    SRVGRP = ORA_GRP1
    SRVID = 20
    RESTART = N
univp_server
    SRVGRP = ORA_GRP2
    SRVID = 20
    RESTART = N
```

オブジェクト状態管理によるアプリケーションのスケールング

「第 1 章 CORBA サーバ・アプリケーションの概念」で説明したように、オブジェクト状態管理は大規模なクライアント / サーバ・システムでは根本的に重要な検討事項です。それは、こうしたシステムでは最適なスループット

および応答時間を実現することが重要であるためです。この節では、BEA Tuxedo サーバ・アプリケーションによって管理されるオブジェクトのスケラビリティを高めるためにオブジェクト状態管理を使用する方法を、Production サンプル・アプリケーションの Registrar および Teller オブジェクトを例にして説明します。

次のテーブルは、使用する BEA Tuxedo アプリケーションのスケラビリティを大きく高めるために、BEA Tuxedo システムでサポートされているオブジェクト状態管理モデルを使用する方法をまとめたものです。

状態モデル	スケラビリティを達成するための使用法
メソッド・バウンド	<p>メソッド・バウンド・オブジェクトは、クライアントによって呼び出されている間だけ、マシンのメモリへ格納されます。呼び出しが完了すると、オブジェクトは非活性化され、そのオブジェクトの状態データはすべて、メモリからフラッシュされます。</p> <p>メソッド・バウンド・オブジェクトは、使用するアプリケーションに状態を持たないサーバ・モデルを作成する場合に使用できます。こうすれば、アプリケーションで数千のオブジェクトを管理できます。クライアント・アプリケーションのビューからは、すべてのオブジェクトがサービス要求に利用可能です。しかし、サーバ・アプリケーションがオブジェクトをメモリにマッピングするのはクライアントによってオブジェクトが呼び出されている間だけなので、任意の時点でメモリ内にあるオブジェクトのうち、サーバ・アプリケーションによって管理されているものは比較的少数にとどまります。</p> <p>メソッド・バウンド・オブジェクトは、このマニュアルでは状態を持たないオブジェクトともいいます。</p>

状態モデル	スケラビリティを達成するための使用法
プロセス・バウンド	<p>プロセス・バウンド・オブジェクトは、最初に呼び出された時点から、自身が実行されているサーバ・プロセスがシャットダウンされるまでメモリ内にとどまります。使用するアプリケーションで支障がなければ、大量の状態データを伴うプロセス・バウンド・オブジェクトは、複数のクライアント呼び出しに対応するためにメモリ内にとどまることができます。このようにすることで、システムのリソースは、クライアント呼び出しごとにオブジェクトの状態データを読み書きすることから解放されます。</p> <p>プロセス・バウンド・オブジェクトは、このマニュアルでは状態を持つオブジェクトともいいます。トランザクション・バウンド・オブジェクトも状態を持つと見なされる点に注意してください。これは、トランザクションのスコープ内であれば、自身への呼び出しの間はメモリ内にとどまることが可能なためです。</p>

スケラビリティによる性能向上を達成するには、Production サーバ・アプリケーションの Registrar および Teller オブジェクトを、method 活性化方針を持つようにコンフィギュレーションする必要があります。これら 2 つのオブジェクトに method 活性化方針を割り当てた結果、振る舞いに次の変化がみられます。

- これらのオブジェクトが呼び出される時は常に、適切なサーバ・グループ内の BEA Tuxedo ドメインによってインスタンス化されます。
- 呼び出しが完了すると、BEA Tuxedo ドメインによりこれらのオブジェクトは非活性化されます。

Basic サンプル・アプリケーションから Wrapper サンプル・アプリケーションを通じて、Registrar オブジェクトはプロセス・バウンドです。このオブジェクトへのすべてのクライアント要求は、マシンのメモリにある同じオブジェクトのインスタンスへ送られます。Basic サンプル・アプリケーションの設計は、すべての小規模なデプロイメントに適しています。しかし、クライアント・アプリケーションの要求が増すにつれて、Registrar オブジェクト上のクライアント要求はキューに入れられるようになり、このため応答時間が遅くなります。

しかし、Registrar および Teller オブジェクトが状態を持たず、これらのオブジェクトを管理するサーバ・プロセスが複製されていれば、これらのオブジェクトはクライアント要求の並行処理に利用可能になります。これらのオブジェクトで同時に処理できるクライアント要求の数に対する唯一の制限は、これらのオブジェクトをインスタンス化できる利用可能なサーバ・プロセスの数です。したがって、これらの状態を持たないオブジェクトはマシン・リソースのより効率的な使用と、クライアント応答時間の短縮を実現します。

最も重要なのは、BEA Tuxedo システムが Registrar および Teller オブジェクトのコピーを、両オブジェクトごとに複製されたサーバ・プロセス内でインスタンス化できるようにするためには、これらのオブジェクトのコピーが一意である必要があるということです。これらのオブジェクトの各インスタンスを一意にするために、オブジェクトのファクトリでは一意のオブジェクト ID をオブジェクトに割り当てる必要があります。このことも含めて、これら 2 つのオブジェクトに関する設計上の検討事項については、第 8 章の 22 ページ「Registrar および Teller オブジェクトに関する設計上の追加考慮事項」を参照してください。

ファクトリ・ベース・ルーティング

ファクトリ・ベース・ルーティングは、特定のサーバ・グループへクライアント要求を送信する方法を提供する強力な機能です。ファクトリ・ベース・ルーティングを使用すれば、アプリケーションの処理負荷を複数のマシンに分散できます。これは、指定されたオブジェクトがインスタンス化されるグループ、すなわちマシンを決定できるからです。

ファクトリ・ベース・ルーティングを使用すれば、BEA Tuxedo システムのさまざまなロード・バランシング機能およびスケラビリティ機能を拡張することができます。Production サンプル・アプリケーションの場合、ファクトリ・ベース・ルーティングを使用すれば、学生のサブセットの 1 つを登録する要求を 1 台のマシンへ送信し、別のサブセットに関する要求は別のマシンへ送信することができます。使用するアプリケーションの処理能力を増やすためにマシンを追加しても、BEA Tuxedo システムではアプリケーションのファクトリ・ベース・ルーティングを簡単に変更してマシンの追加に対応できます。

ファクトリ・ベース・ルーティングの第一の利点は、デプロイメント環境の拡大に対応して、アプリケーション、特にインターフェイスの呼び出し機能をスケール・アップするための簡単な方法が提供されることです。アプリケーションのデプロイメント範囲を追加マシンにも広げることは、厳密には管理用の機能であり、アプリケーションの再コーディングや再ビルドは不要です。

クライアント / サーバ・アプリケーションにファクトリ・ベース・ルーティングをインプリメントする際に設計上最も重要な検討事項は、ルーティングのベースとなる値の選択です。以下の節では、ファクトリ・ベース・ルーティングの機能について、Production サンプル・アプリケーションを使用して説明します。Production サンプル・アプリケーションは、ファクトリ・ベース・ルーティングを次のように使用します。

- Registrar オブジェクトへのクライアント・アプリケーション要求は、学生 ID に基づいてルーティングされます。つまり、学生のサブセットの 1 つに関する要求が 1 つのグループに送られ、別のサブセットに関する要求は別のグループに送られます。
- Teller オブジェクトへの要求は、口座番号に基づいてルーティングされます。つまり、口座のサブセットの 1 つに関する要求が 1 つのグループに送られ、別のサブセットに関する要求は別のグループに送られます。

ファクトリ・ベース・ルーティングのしくみ

ファクトリでファクトリ・ベース・ルーティングをインプリメントする際には、ファクトリがオブジェクト・リファレンスを作成する方法が変更されません。すべてのオブジェクト・リファレンスはグループ ID を含み、デフォルトのグループ ID はオブジェクト・リファレンスを作成するファクトリと同じになります。しかし、ファクトリ・ベース・ルーティングを使用する場合、ファクトリはグループ ID を決定するルーティング基準も含めてオブジェクト・リファレンスを作成します。その後、クライアント・アプリケーションがこうしたオブジェクト・リファレンスを使用して呼び出しを送信すると、BEA Tuxedo システムによって、要求はオブジェクト・リファレンスに指定されたグループ ID ヘルルーティングされます。ここでは、オブジェクト・リファレンスに関するグループ ID がどのようにして生成されるかを説明します。

ファクトリ・ベース・ルーティングをインプリメントするには、次の項目を調整する必要があります。

- UBBCONFIG ファイルの INTERFACES および ROUTING セクションのデータ。
- UBBCONFIG ファイルでコンフィギュレーションされているグループ、マシンおよびデータベース。
- ファクトリでルーティング基準が指定される方法。ファクトリのためのインターフェイス定義では、グループ ID の決定に使用されるルーティング基準を表すパラメータを指定する必要があります。

調整が必要なデータについて説明するために、以降の 2 つの節では、UBBCONFIG ファイルでのファクトリ・ベース・ルーティングのコンフィギュレーションと、ファクトリでのファクトリ・ベース・ルーティングのインプリメンテーションを説明します。

UBBCONFIG ファイルでのファクトリ・ベース・ルーティングのコンフィギュレーション

要求がルーティングされる各インターフェイスについて、UBBCONFIG ファイルに次の情報を記述する必要があります。

- ルーティング基準のデータの詳細
- 各基準ごとの、特定のサーバ・グループへルーティングするための値

ファクトリ・ベース・ルーティングをコンフィギュレーションするには、UBBCONFIG ファイルの INTERFACES および ROUTING セクションで次のデータを指定するほか、グループおよびマシンを識別する方法を指定する必要があります。

1. INTERFACES セクションには、ファクトリ・ベース・ルーティングを有効にするインターフェイスの名前の一覧が示されます。各インターフェイスについて、このセクションでインターフェイスのルーティングの基準の種類を指定します。このセクションでは、次の例のように、FACTORYROUTING という識別子でルーティング基準を指定します。

```
INTERFACES
  "IDL:beasys.com/UniversityP/Registrar:1.0"
    FACTORYROUTING = STU_ID
  "IDL:beasys.com/BillingP/Teller:1.0"
    FACTORYROUTING = ACT_NUM
```

上の例では、Production サンプルでファクトリ・ベース・ルーティングが使用される 2 つのインターフェイスの完全修飾されたインターフェイス名を示します。FACTORYROUTING 識別子は、ルーティング値の名前を指定します。値はそれぞれ、STU_ID および ACT_NUM です。

2. ROUTING セクションでは、ルーティング値ごとに以下のデータを指定します。
 - TYPE パラメータ。ルーティングの種類を指定します。Production サンプルでは、ルーティングのタイプはファクトリ・ベース・ルーティングです。したがって、このパラメータは、FACTORY と定義します。
 - FIELD パラメータ。ファクトリがルーティング値に挿入する名前を指定します。Production サンプルでは、フィールドのパラメータはそれぞれ student_id および account_number です。
 - FIELDTYPE パラメータ。ルーティング値のデータ型を指定します。Production サンプルでは、student_id および account_number のフィールド・タイプは long です。
 - RANGES パラメータ。各グループにルーティングされる値を指定します。

次に、Production サンプル・アプリケーションで使用する UBBCONFIG ファイルの ROUTING セクションの例を示します。

```
ROUTING
STU_ID
  FIELD      = "student_id"
  TYPE       = FACTORY
  FIELDTYPE  = LONG
  RANGES    = "100001-100005:ORA_GRP1,100006-100010:ORA_GRP2"
ACT_NUM
  FIELD      = "account_number"
  TYPE       = FACTORY
  FIELDTYPE  = LONG
  RANGES    = "200010-200014:APP_GRP1,200015-200019:APP_GRP2"
```

上の例では、一方の範囲に入る ID を持った学生についての Registrar オブジェクト・リファレンスは一方のサーバ・グループにルーティングされ、もう一方の範囲に入る ID を持った学生についての Registrar オブジェクト・リファレンスは他方のグループにルーティングされることが示されています。同様に、ある 1 つの範囲の口座に対する Teller オブジェクト・リファレンスは、ある 1 つのサーバ・グループに

ルーティングされ、別の範囲の口座に対する Teller オブジェクト・リファレンスは、別のグループにルーティングされます。

3. UBBCONFIG ファイルの ROUTING セクションにある RANGES 識別子で指定されたグループを識別およびコンフィギュレーションする必要があります。たとえば、Production サンプルでは 4 つのグループが指定されています。APP_GRP1、APP_GRP2、ORA_GRP1、および ORA_GRP2 です。これらのグループをコンフィギュレーションし、グループが実行されるマシンを識別する必要があります。

次の例は、Production サンプルの UBBCONFIG ファイルの GROUPS セクションを示します。このセクションで、ORA_GRP1 および ORA_GRP2 グループがコンフィギュレーションされています。GROUPS セクションに列挙されている名前が、ROUTING セクションに指定されているグループ名とどのように一致するか注目してください。この点が、ファクトリベース・ルーティングの正常な動作に不可欠です。さらに、アプリケーションでグループをコンフィギュレーションする方法に関する何らかの変更も、ROUTING セクションに反映される必要があります。BEA Tuxedo ソフトウェアとともにパッケージされている Production サンプルは 1 台のマシンでだけ実行できるようにコンフィギュレーションされている点に注意してください。ただし、このアプリケーションを複数のマシンで実行できるようにコンフィギュレーションすることは簡単です。

*GROUPS

```
APP_GRP1
  LMID          = SITE1
  GRPNO         = 2
  TMSNAME       = TMS
APP_GRP2
  LMID          = SITE1
  GRPNO         = 3
  TMSNAME       = TMS
ORA_GRP1
  LMID          = SITE1
  GRPNO         = 4
  OPENINFO      = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
  CLOSEINFO     = ""
  TMSNAME       = "TMS_ORA"
ORA_GRP2
  LMID          = SITE1
  GRPNO         = 5
  OPENINFO      = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
  CLOSEINFO     = ""
  TMSNAME       = "TMS_ORA"
```

ファクトリでのファクトリ・ベース・ルーティングのインプリメンテーション

ファクトリは、`TP::create_object_reference()` オペレーションへの呼び出しがインプリメントされている方法によってファクトリ・ベース・ルーティングをインプリメントします。このオペレーションには、次の C++ バインディングがあります。

```
CORBA::Object_ptr TP::create_object_reference (
    const char* interfaceName,
    const PortableServer::oid &stroid,
    CORBA::NVlist_ptr criteria);
```

このオペレーションに対する 3 番目のパラメータである `criteria` は、ファクトリ・ベース・ルーティングに使用される、名前の付いた値を指定します。このため、ファクトリでファクトリ・ベース・ルーティングをインプリメントする機能は、`NVlist` をビルドすることにあります。

前述のように、Production サンプル・アプリケーションの `RegistrarFactory` オブジェクトは、値 `STU_ID` を指定します。この値は、`UBBCONFIG` ファイルの次の項目と正確に一致する必要があります。

- `INTERFACES` セクションの `FACTORYROUTING` 識別子で指定される、ルーティング名、タイプ、および使用できる値
- `ROUTING` セクションで指定される、ルーティング基準名、フィールド、およびフィールド型

`RegistrarFactory` オブジェクトは、次のコードを使用して `NVlist` に学生 ID を挿入します。

```
// put the student id (which is the routing criteria)
// into a CORBA NVList:
CORBA::NVList_var v_criteria;
TP::orb()->create_list(1, v_criteria.out());
CORBA::Any any;
any <<= (CORBA::Long)student;
v_criteria->add_value("student_id", any, 0);
```

`RegistrarFactory` オブジェクトは次のようにして `TP::create_object_reference()` オペレーションを呼び出し、前のコード例で作成された `NVlist` を渡します。

```
// ルーティング基準を使用して、registrar オブジェクト・
// リファレンスを作成
CORBA::Object_var v_reg_oref =
    TP::create_object_reference(
        UniversityP::_tc_Registrar->id(),
        object_id,
        v_criteria.in()
    );
```

Production サンプル・アプリケーションはまた、TellerFactory オブジェクトでもファクトリ・ベース・ルーティングを使用し、口座番号に基づいて Teller オブジェクトがインスタンス化されるべきグループを決定します。

注記 指定されたインターフェイスおよび OID を持つ 1 つのオブジェクトが 2 つの異なるグループで同時に活性化されることも、それら 2 つのグループに同じオブジェクト・インプリメンテーションがある場合には起こりえます。しかし、使用するファクトリで一意的な OID が生成されていれば、こうした状況はほぼありません。指定されたインターフェイス名および OID の 1 つのオブジェクトがドメイン内で一度に 1 つだけ利用可能であることを保証する必要がある場合は、次のいずれかを行います。ファクトリ・ベース・ルーティングを使用して、特定の OID のオブジェクトが常に同じグループにルーティングされるようにするか、指定されるオブジェクト・インプリメンテーションが 1 つのグループのみに存在するようにドメインをコンフィグレーションする。そうすれば、指定されたインターフェイス名および OID を持つ 1 つのオブジェクトに複数のクライアントがオブジェクト・リファレンスをした場合でも、そのリファレンスは常に同じオブジェクト・インスタンスヘルレーティングされることとなります。

オブジェクトの OID によるルーティングを利用可能にするには、TP::create_object_reference() オペレーションで OID をルーティング基準に指定してから、UBBCONFIG ファイルをそれに合わせて設定します。

実行時の処理

ファクトリでファクトリ・ベース・ルーティングをインプリメントするとき、BEA Tuxedo システムによってオブジェクト・リファレンスが生成されます。次の例は、ファクトリ・ベース・ルーティングがインプリメントされているときにクライアント・アプリケーションが `Registrar` オブジェクトへのオブジェクト・リファレンスを取得する方法を示します。

1. クライアント・アプリケーションが、`RegistrarFactory` オブジェクトを呼び出し、`Registrar` オブジェクトへのリファレンスを要求します。要求には学生 ID が含まれます。
2. `RegistrarFactory` は、`NVlist` に学生 ID を挿入します。これは、ルーティング基準として使用されます。
3. `RegistrarFactory` は、`TP::create_object_reference()` オペレーションを呼び出し、`Registrar` インターフェイス名、一意の OID、および `NVlist` を渡します。
4. BEA Tuxedo システムはルーティング・テーブルの内容を `NVlist` の値と比較して、グループ ID を判別します。
5. BEA Tuxedo システムは、オブジェクト・リファレンスにグループ ID を挿入します。

続いてクライアント・アプリケーションがオブジェクト・リファレンスを使用してオブジェクト呼び出しを実行した場合、BEA Tuxedo システムはその要求を、オブジェクト・リファレンスで指定されたグループヘルルーティングします。

注記 `Process-Entity` デザイン・パターンを使用する場合は、ファクトリ・ベース・ルーティングのインプリメンテーションに注意してください。オブジェクトは、グループのデータベースに格納されているエンティティしか処理できません。

Registrar および Teller オブジェクトに関する設計上の追加考慮事項

Registrar および Teller オブジェクトの設計に影響する主な考慮事項には、以下のものがあります。

- Registrar および Teller オブジェクトが Production のデプロイメント環境、つまり、複数の複製されたサーバ・プロセスおよび複数のグループで適切に機能するようにする方法。University および Billing の各サーバ・プロセスが複製されている場合の設計では、これら 2 つのオブジェクトをどのようにインスタンス化するかを考慮する必要があります。
- Production の BEA Tuxedo ドメインにある 2 つのサーバ・グループがそれぞれ別のデータベースを扱う場合に、特定の学生についての登録用のオペレーションおよび課金用のオペレーションに関するクライアント要求が適切なサーバ・グループへ送られるようにする方法。

これらの考慮事項の主要な意味は、これらのオブジェクトが次の項目を満たす必要があるということです。

- 一意なオブジェクト ID (OID) を持っている。
- メソッド・バウンドである。つまり、これらのオブジェクトには `method` 活性化方針が割り当てられている。

以降の節では、これらの検討事項およびその内容について詳しく説明します。

Registrar および Teller オブジェクトのインスタンス化

Production サンプル・アプリケーションの前に扱った University サーバ・アプリケーションでは、Registrar および Teller オブジェクトの実行時の振る舞いは簡単なものでした。

- 各オブジェクトは、プロセス・バウンドでした。つまり、最初に呼び出されたときに活性化され、そのオブジェクトが実行されているサーバ・プロセスがシャットダウンされるまで、メモリ内に残っていました。

- BEA Tuxedo ドメインで実行されるサーバ・グループは1つしかなく、グループ内では University および Billing のサーバ・プロセスは1つしかなかったため、すべてのクライアント要求は、同じオブジェクトに転送されました。複数のクライアント要求が BEA Tuxedo ドメインに到達する一方で、これらの各オブジェクトが処理するクライアントは一度に1つずつでした。
- 各オブジェクトのインスタンスは、それらが実行されているサーバ・プロセスに1つしかなかったため、どちらのオブジェクトにも一意の OID は不要でした。各オブジェクトの OID が指定するのは、インターフェイス・リポジトリ ID のみでした。

しかし、University および Billing サーバ・プロセスが複製されると、BEA Tuxedo ドメインには Registrar および Teller オブジェクトの複数のインスタンスを区別する方法が必要になります。つまり、1つのグループで2つの University サーバ・プロセスが実行中である場合、BEA Tuxedo ドメインには、いわば、1番目の University サーバ・プロセスで実行中の Registrar オブジェクトと、2番目の University サーバ・プロセスで実行中の Registrar オブジェクトを区別するための方法が必要になります。

これらのオブジェクトの複数のインスタンスを区別する機能を BEA Tuxedo ドメインに与える方法は、オブジェクトの各インスタンスを一意にすることです。

各 Registrar および Teller オブジェクトを一意のものにするには、これらのオブジェクトのファクトリで、これらに対するオブジェクト・リファレンスを作成する方法を変更しなければなりません。たとえば Basic サンプル・オブジェクトでは、RegistrarFactory オブジェクトが Registrar オブジェクトへのオブジェクト・リファレンスを作成すると、`TP::create_object_reference()` オペレーションが、文字列 `registrar` のみからなる OID を指定していました。しかし、Production サンプル・アプリケーションでは、同じ `TP::create_object_reference()` オペレーションが、生成された一意の OID を使用します。

Registrar および Teller オブジェクトのそれぞれに一意な OID を付与した結果、これらのオブジェクトの複数のインスタンスが、BEA Tuxedo ドメインの中で同時に実行できるようになりました。この特性は状態を持たないオブジェクト・モデルに典型的なもので、BEA Tuxedo ドメインのスケラビリティを高めつつ、高い性能を提供する方法の例でもあります。

そして最後に、一意な Registrar および Teller オブジェクトはクライアント要求ごとにメモリに格納される必要があるため、これらのオブジェクトへの呼び出しが完了したときには、オブジェクトの状態がメモリ上でアイドル状態となって残らないように、オブジェクトを非活性化することが非常に重要になります。Production サーバ・アプリケーションでは、ICF ファイルでこれら 2 つのオブジェクトに method 活性化方針を割り当てることで、この問題を解決しています。

適切なサーバ・グループで学生の登録が発生するようにする方法

複製されたサーバ・グループを使用することでスケラビリティについて得られる最大の利点は、複数のマシンに処理を分散できることです。しかし、University サンプル・アプリケーションの場合のように、アプリケーションがデータベースと対話する場合は、これら複数のサーバ・グループがデータベースとの対話に与える影響を考慮することが重要です。

多くの場合、デプロイされているマシンに、データベースは 1 つずつ関連付けられています。サーバ・アプリケーションが複数台のマシンに分散されている場合は、データベースをどのようにセットアップするかを検討する必要があります。

この章で説明しているように、Production サンプル・アプリケーションでは、2 つのデータベースを使用します。しかし、このアプリケーションは、簡単にそれ以上のデータベースに対応できるようコンフィギュレーションできます。使用するデータベースの数は、システム管理者が決定してください。

Production サンプル・アプリケーションでは、学生および口座の情報は、2 つのデータベース間に分割されますが、コース情報は同一です。コース情報は、コース登録用途では読み取り専用となっているため、両方のデータベースのコース情報が同一であることは問題にはなりません。一方で、学生情報および口座情報については読み書きが行われます。複数のデータベースが学生および口座についても同一のデータを格納するとしたら（つまり、データベースが分割されていない場合）、アプリケーションでは、学生情報または口座情報が変更されるたびにデータベース全体にわたって学生および口座の情報の更新を同期する処理のオーバーヘッドに対処する必要が生じるでしょう。

Production サンプル・アプリケーションは、ファクトリ・ベース・ルーティングによって、1 台のマシンに要求の 1 セットを、別のマシンに要求の別の 1 セットを送信します。先に説明したように、ファクトリ・ベース・ルーティングは、Registrar オブジェクトへのリファレンスが作成されている方法によって RegistrarFactory オブジェクトにインプリメントされています。

たとえば、クライアント・アプリケーションが RegistrarFactory オブジェクトに要求を送信して、Registrar オブジェクトのオブジェクト・リファレンスを取得した場合、クライアント・アプリケーションはその要求に学生 ID を含めます。クライアント・アプリケーションは、RegistrarFactory オブジェクトが返すオブジェクト・リファレンスを使用して、その後の Registrar オブジェクトへのすべての呼び出しを、特定の学生のために行う必要があります。ファクトリによって返されたオブジェクト・リファレンスは、グループ固有のものだからです。したがって、たとえばクライアント・アプリケーションがその後、Registrar オブジェクトに対して `get_student_details()` オペレーションを呼び出すと、クライアント・アプリケーションでは、Registrar オブジェクトが、その学生のデータを格納しているデータベースと関連付けられたサーバ・グループにおいて、確実に活性化されます。この機能を示すために、Production サンプル・アプリケーションにインプリメントされている以下の実行シナリオを検討してみます。

1. クライアント・アプリケーションは、RegistrarFactory オブジェクトの `find_registrar()` オペレーションを呼び出します。この要求には学生 ID 1000003 が含まれます。
2. BEA Tuxedo ドメインは、クライアント要求を任意の RegistrarFactory オブジェクトにルーティングします。
3. RegistrarFactory オブジェクトは、UBBCONFIG ファイルのルーティング情報に基づいて、学生 ID を使用して ORA_GRP1 の Registrar オブジェクトへのオブジェクト・リファレンスを作成し、このオブジェクト・リファレンスをクライアント・アプリケーションに返します。
4. クライアント・アプリケーションは、Registrar オブジェクトの `register_for_courses()` オペレーションを呼び出します。

5. BEA Tuxedo ドメインは、クライアント要求を受信してから、それをオブジェクト・リファレンスで指定されているサーバ・グループヘルディングします。この場合、クライアント要求は Production マシン 1 にある ORA_GRP1 の University サーバ・プロセスに送信されます。
6. University サーバ・プロセスは、Registrar オブジェクトをインスタンス化し、それにクライアント呼び出しを送信します。

以上のシナリオの RegistrarFactory オブジェクトがクライアント・アプリケーションに返すのは、ORA_GRP1 でだけインスタンス化できる Registrar オブジェクトへの一意なリファレンスです。ORA_GRP1 は Production マシン 1 で実行されていて、学生 ID が 100001 ~ 100005 の範囲の学生に関するデータが格納されているデータベースを利用します。このため、続けてクライアント・アプリケーションが学生に関する Registrar オブジェクトへの要求を送信したとき、Registrar オブジェクトは適切なデータベースとやり取りができます。

Teller オブジェクトが適切なサーバ・グループでインスタンス化されるようにする方法

Registrar オブジェクトで Teller オブジェクトが必要になると、Registrar オブジェクトは TellerFactory オブジェクトを呼び出します。このとき、University Server オブジェクトでキャッシュされている TellerFactory オブジェクト・リファレンスが使用されます。詳細については、第 7 章の 13 ページ「Teller オブジェクトへの要求の送信」を参照してください。

しかし、TellerFactory オブジェクトではファクトリ・ベース・ルーティングが使用されているため、Registrar オブジェクトは Teller オブジェクトへのリファレンス要求時に、学生の口座番号を渡します。このようにして TellerFactory オブジェクトは、正しいデータベースを備えたグループ内の Teller オブジェクトへのリファレンスを作成します。

注記 Production サンプル・アプリケーションが適正に動作するためには、システム管理者がサーバ・グループとデータベースのコンフィギュレーションを正しく行うことが不可欠です。具体的には、システム管理者はルーティング・テーブルで指定されたルーティング基準と、これらの基準を使用した要求のルーティング先となるデータベースを、確実に一致させる必要があります。Production サンプルを例にす

ると、指定されたグループにあるデータベースは、そのグループにルーティングされた要求に正しく対応する学生情報および口座情報を含んでいる必要があります。

Production サーバ・アプリケーションをさらにスケーリングする方法

将来的には、Production サンプル・アプリケーションのシステム管理者が BEA Tuxedo ドメインの容量を増やす必要を感じる場合もあります。たとえば、University で学生数が大幅に増加した場合や、7つのキャンパスがある大学の全体のコース登録処理を扱えるように Production アプリケーションをスケール・アップする場合などが考えられます。これを行うのに、アプリケーションを修正したり再ビルドしたりする必要はありません。

継続的に容量を追加するために、システム管理者には以下のツールが用意されています。

- Production サンプル・アプリケーションを複数のマシンにある複数のサーバ・グループに複製する場合。

UBBCONFIG ファイルを変更する必要があります。サーバ・プロセスが実行されるグループを追加し、グループでどのサーバ・プロセスを実行するか指定してから、どのマシンで実行するかを指定してください。

- ファクトリ・ベース・ルーティングのテーブルを修正します。

たとえば、この章で示したように2つのグループヘルルーティングする代わりに、システム管理者が UBBCONFIG ファイルでルーティング・ルールを変更することで、BEA Tuxedo ドメインに追加された新しいグループの間でアプリケーションの処理をさらに分離することが可能です。ルーティング・テーブルへのすべての変更は、UBBCONFIG ファイルでコンフィギュレーションされているサーバ・グループおよびマシンに対するすべての変更および追加と整合している必要があります。

注記 データベースを使用するアプリケーションの容量を増やす場合、特にファクトリ・ベース・ルーティングを使用しているときには、データベースの設定に与える影響を検討する必要があります。たとえば、Production サンプル・アプリケーションが 6 台のマシン間に分散している場合は、UBBCONFIG ファイルのルーティング・テーブルに基づいて、適切に各マシンのデータベースを設定する必要があります。

状態を持たないオブジェクトまたは状態を持つオブジェクトの選択

一般に、状態を持たないオブジェクトをインプリメントする際のコストと、状態を持つオブジェクトをインプリメントする際のコストを比較する必要があります。

オブジェクトをその永続状態で初期化するコストが高い場合（これは、たとえば、そのオブジェクトのデータが大きな領域を占めている場合や、永続状態の場所が、それを活性化するサーバントから大幅に離れたディスク上である場合ですが）、たとえオブジェクトが会話の間アイドル状態であるとしても、オブジェクトが状態を持たないようにしておく方が合理的です。オブジェクトを活性化されたままにしておくコストが、マシンのリソース使用率との関連で高額になる場合は、そのようなオブジェクトは状態を持たないようにするほうが合理的です。

オブジェクトの状態を、アプリケーションに合わせて効率的かつ適切な方法で管理することにより、アプリケーションの能力を最大限に高めて、多数のオブジェクトを使用する多数のクライアント・アプリケーションを同時にサポートできます。一般には、こうした目的のために、オブジェクトに対して `method` 活性化方針を割り当てます。こうすることによって、アイドル状態のオブジェクト・インスタンスが非活性化され、マシンのリソースをほかのオブジェクト・インスタンスに割り当てられるようになります。ただし、使用するアプリケーション固有の特性および要件は、アプリケーションごとに異なります。

注記 BEA Tuxedo リリース 8.0 では、性能の拡張として、パラレル・オブジェクトのサポートが提供されています。この機能を利用すると、特定アプリケーションのすべてのビジネス・オブジェクトを状態を持たないオブジェクトにすることができます。詳細については、『BEA Tuxedo CORBA プログラミング・リファレンス』の「第 3 章 TP フレームワーク」を参照してください。相互参照が当たっていません (2003/02/27)

状態を持たないオブジェクトが必要な場合

サーバ・リソースはオブジェクトがアイドル状態のときには絶対に使用されないため、状態を持たないオブジェクトは一般に、サーバ・リソースの性能を高め、最適な使い方を実現します。状態を持たないオブジェクトは、一般に、サーバ・アプリケーションのインプリメンテーションに適した手法です。状態を持たないオブジェクトは、特に以下の状況に適しています。

- クライアント・アプリケーションは、オブジェクトへの呼び出しの間のユーザ入力を通常は待機する。
- クライアント要求はサーバ・アプリケーションが必要とするデータを通常はすべて含み、サーバはそのデータのみを使用してクライアント要求を処理できる。
- オブジェクトのアクセス・レートはたいへん高いが、特定のクライアント・アプリケーションからのアクセス・レートが低い場合。

オブジェクトを状態を持たないようにすることで、サーバ・アプリケーションのリソースがクライアント・アプリケーションからの入力の待機のために不定期に長時間拘束されないようにできます。

状態を持たないオブジェクト・モデルを使用するアプリケーションでは、次の特性に注意してください。

- 呼び出しとそれに関連する情報は、サーバ・アプリケーションがクライアント要求を実行し終わった後は保持されません。
- 着信するクライアント要求は、最初に利用可能なサーバ・プロセスに送信されます。要求が終了した後は、アプリケーションの状態が消滅して、サーバ・アプリケーションは別のクライアント・アプリケーション要求で利用可能になります。

- オブジェクトの永続状態の情報は、サーバ・プロセス外に存在します。このオブジェクトが呼び出されるたびに、永続状態がメモリに読み込まれます。
- BEA Tuxedo ドメインは、クライアント・アプリケーションから 1 つのオブジェクトへの連続した要求を別のサーバ・プロセスへ割り当てることがあります。
- 通常、状態を持たないオブジェクトを実行しているマシンの全体的なシステム性能が向上します。

状態を持つオブジェクトが必要な場合

状態を持つオブジェクトは、いったん活性化されると、オブジェクトが存在しているプロセスがシャットダウンされたり、オブジェクトが活性化されているトランザクションが完了したりといった、特定のイベントが発生するまで、メモリ内にとどまります。

状態を持つオブジェクトは、一般に以下の状況に適しています。

- 1 つのオブジェクトが、多数のクライアント・アプリケーションによってたいへん頻繁に利用される場合。これには、ファクトリのように寿命が長く、よく知られたオブジェクトが該当します。サーバ・アプリケーションがこれらのオブジェクトを活性化されたままにしていると、クライアント・アプリケーションがそれらにアクセスする際の応答時間は通常、最短化されません。活性化されたオブジェクトは多くのクライアント・アプリケーションに共有されるので、この型のオブジェクトでメモリにあるものは比較的少数です。

注記 プロセス・オブジェクトをトランザクションにどのように関与させるかについては、注意深く検討してください。トランザクションに關与しているすべてのオブジェクトは、ほかのクライアント・アプリケーションまたはオブジェクトから呼び出せません。プロセス・オブジェクトは多数のクライアント・アプリケーションによって利用されるので、トランザクションに対して頻繁に、または長期間関与すると、問題の原因になる可能性があります。

- トランザクションを完了するために 1 つのオブジェクト上でクライアント・アプリケーションがオペレーションを連続して呼び出す必要がある場合、および、こうした呼び出しの間にユーザからの入力を待機する際

にクライアント・アプリケーションがアイドル状態でない場合。この場合、呼び出しの間にオブジェクトが非活性化されると、各呼び出しの間に状態の読み書きが発生するために、応答時間の遅延が発生します。こうした振る舞いは、トランザクションには適していません。応答時間を改善するには、サーバ・リソースの保持を犠牲にするという方法が可能です。

状態を持つオブジェクトの以下の振る舞いに注意してください。

- 状態の情報はサーバの呼び出しの間に保持されます。通常は、指定された時間にわたって、指定されたクライアント・アプリケーションに専用のサーバントが存在します。
- データはクライアント・アプリケーションとサーバ・アプリケーションの間で送受信されますが、サーバ・プロセスにより追加のコンテキストまたはアプリケーション状態情報がメモリ内に保持されます。
- 1つまたは複数の状態を持つオブジェクトが大量のマシン・リソースを使用している場合、状態を持つオブジェクトと関連付けられていないタスクおよびプロセスを扱うときのサーバの性能は、状態を持たないサーバ・モデルの場合よりも悪化します。

たとえば、1つのオブジェクトがデータベースにロックを保持し、大量のデータをメモリにキャッシュしている場合、その状態を持つオブジェクトによって使用されているデータベースおよびメモリは、トランザクションが完了するまでほかのオブジェクトでは利用できません。

索引

記号

A

- ACID 特性 6-2
- activate_object メソッド 4-39
- activate_object() オペレーション
 - 事前活性化オブジェクト 3-21
 - 例 3-15
 - 例外 2-27
- adm ディレクトリ 4-57
- always トランザクション方針 6-14
 - 例 7-15
- application_responsibility() オペレーション 2-35
- AUTOTRANS
 - トランザクション・オブジェクトを参照 6-14

B

- BAD_OPERATION 2-27
- Basic University サンプル
 - ICF ファイル 3-14
 - OMG IDL 3-3
 - 永続状態の処理 3-14
 - オブジェクト状態の管理 3-12
 - 概要 3-2
 - 設計に関する考慮事項 3-9
 - デザイン・パターンの使用 3-17
- BEA Tuxedo サーバ・アプリケーション
 - BEA Tuxedo ドメインでの使用 7-2
 - 呼び出すオブジェクトの設計 7-3
- BEA Tuxedo サービス
 - CORBA アプリケーションからの呼び

- 出し 7-3
- バッファ型の選択 7-5
- Billing サーバ・アプリケーション
 - University サンプル 7-12
- build コマンド
 - buildobjclient 4-34
 - buildobjserver 4-34
- buildobjclient コマンド 4-34
 - コンパイラ設定 4-38
 - スレッド・サポート・ライブラリ 4-38
- buildobjserver コマンド 4-34
 - b オプション 4-37, 4-38
 - t オプション 4-39, 4-64
 - コンパイラ設定 4-36
 - スレッド・ライブラリ 4-36
 - スレッドをサポートするための修正 4-36
 - マルチスレッド・サポートの指定 4-64

C

- CONCURR_STRATEGY パラメータ 4-65
- CORBA インターフェイス 4-46
- CORBA オブジェクト
 - オブジェクトを参照 1-5
- CORBA サーバ・アプリケーション
 - セキュリティ 5-2
 - トランザクション 6-5
- CORBA::Current 4-31
- CosTransactions::Control オブジェクト 4-31
- CosTransactions::Current オブジェクト 4-31
- create_active_object_reference() オペレーション 3-20

create_object_reference() オペレーション
ルーティング基準の指定 8-19
例 2-9

create_servant() オペレーション
OBJECT_NOT_EXIST 2-32
例外 2-27

create_servant_with_id メソッド 4-33

Current オブジェクト 4-31
オペレーションとインターフェイス
4-31
ナロー変換 4-31

D

deactivate_object メソッド 4-39

deactivate_object() オペレーション
サーバント・プール 2-35
使用上の制限 2-33
状態の処理 2-33
トランザクション 6-20
例外 2-27

deactivateEnable() オペレーション 3-13
概要 1-17
事前活性化オブジェクト 3-21
例 3-13

E

expected_output ファイル 4-58

F

FML 7-5

FML32 バッファ
割り当て 7-5

FML32 バッファの割り当て 7-5

forward_lower メソッド 4-44

forward_upper メソッド 4-44

I

ICF ファイル 2-9
トランザクション方針の割り当て

6-18

IDL

OMG IDL を参照 1-3

idl コマンド 2-3

IDL コンパイラ 1-4
tie クラスの生成 2-7
使用方法 2-5

ignore トランザクション方針 6-17

IOP リスナ / ハンドラ 8-2

input ファイル 4-57

INVALID_TRANSACTION 例外 6-22

is_reentrant メソッド 4-30

K

Korn シェル 4-48

L

List-Enumerator デザイン・パターン (例)
3-18

List-Enumerator デザイン・パターン 1-31

M

makefile.mk ファイル 4-51

makefile.nt ファイル 4-51

MAXACCESSORS パラメータ 4-64

MAXDISPATCHTHREADS パラメータ
4-37, 4-63
ほかのパラメータへの影響 4-64

MINDISPATCHTHREADS パラメータ
4-63, 4-65

N

never トランザクション方針 6-16

new
C++ ステートメント 1-8

O

OBJ_ADAPTER 例外 6-22

OBJECT_NOT_EXIST 2-27

2-32

OID 3-9

OMG CORBA

仕様 4-31

OMG IDL

Basic University サンプル 3-3

Production University サンプル 8-5

Wrapper University サンプル 7-12

オブジェクトの定義 1-3

オペレーションの定義 1-3

バージョンの不一致 2-32

OMG IDL のコンパイル 2-3

open_xa_rm() オペレーション 6-18

optional トランザクション方針 6-15

Oracle 6-11

ORB Portability 仕様 4-31

ORB::clear_ctx 4-33

ORB::set_ctx 4-33

output ファイル 4-58

P

PortableServer::Current オブジェクト 4-31

Process-Entity デザイン・パターン (例)
3-18

Process-Entity デザイン・パターン 1-30

Production University サンプル

OMG IDL 8-5

UBBCONFIG ファイル 8-9

R

Readme.txt ファイル 4-52

Registrar オブジェクト

Transaction University サンプルでの方
針 6-10

RegistrarFactory オブジェクト 3-9

results ディレクトリ 4-57

runme.cmd ファイル 4-52

runme.ksh ファイル 4-52

S

samplesdb.h 3-16

SECURITY

パラメータ、UBBCONFIG ファイル
5-3

Security University サンプル

OMG IDL 5-6

概要 5-4

設計 5-3

SecurityCurrent オブジェクト 5-4

SecurityLevel1::Current オブジェクト 4-31

SecurityLevel2::Current オブジェクト 4-31

Server クラス 4-37

ServerBase クラス 4-33, 4-37

setenv.cmd ファイル 4-58

setenv.ksh ファイル 4-58

simpapp_mt サンプル・アプリケーション
しくみ 4-43

パーミッションの変更 4-53

ビルドおよび実行 4-46

ファイルのリスト 4-51

simple.idl ファイル 4-52

simple_c.cpp ファイル 4-56

simple_c.h ファイル 4-56

simple_client ファイル 4-57

simple_client.cpp ファイル 4-52

simple_per_object_i.cpp ファイル 4-52

simple_per_object_i.h ファイル 4-52

simple_per_object_server ファイル 4-57

simple_per_request_i.cpp ファイル 4-53

simple_per_request_i.h ファイル 4-53

simple_per_request_server ファイル 4-57

simple_per_request_server.cpp ファイル
4-53

simple_per_request_server.h ファイル 4-53

simple_s.cpp ファイル 4-56

simple_s.h ファイル 4-56

SimplePerRequest サーバ・プロセス 4-44

SimplePerRequestFactory_i

インターフェイス 4-56

インプリメンテーション 4-56

stderr ファイル 4-58

stdout ファイル 4-58

T

thread_initialize メソッド 4-33
thread_macros.cpp ファイル 4-53
thread_macros.h ファイル 4-53
thread_release メソッド 4-34
tie クラス
 生成 2-7
 デレゲーション・ベースのインターフェイス・インプリメンテーションを参照 2-36
TMS 6-11
 Oracle 6-11
 コンフィギュレーション 6-11
 要件 6-11
tmsysevt.dat ファイル 4-59
to_lower メソッド 4-44
to_upper メソッド 4-44
Tobj_ServantBase クラス 4-34
Tobj_ServantBase::_is_reentrant メソッド 4-34
Tobj_serverBase::_add_ref メソッド 4-34
Tobj_serverBase::_remove_ref メソッド 4-34
TobjS_c.h 2-26
TobjServantBase::_is_reentrant メソッド 4-41
TP フレームワーク 4-25
transaction 活性化方針 6-20
Transactions University サンプル
 オブジェクト状態管理 6-9
 概要 6-5
 コンフィギュレーション 6-11
 しくみ 6-7
tuxconfig ファイル 4-59
TUXDIR 4-47, 4-48
Tuxedo
 BEA Tuxedo を参照 7-3
Tuxedo サービスのラッピング
 オブジェクトとして 7-3

U

ubb ファイル 4-59
UBBCONFIG ファイル 4-35, 4-37
 Production University サンプル 8-9
 SECURITY パラメータ 5-3
 概要 2-23
 制御パラメータ 4-35
 設定 4-29
 ファクトリ・ベース・ルーティング 8-16
 例 4-67
ULOG.date ファイル 4-59

W

Wrapper University サンプル
 コンフィギュレーション 7-15
 しくみ 7-11
 設計のまとめ 7-9

X

XA リソース・マネージャ
 Transaction University サンプルでの使用 6-11
 オープン 6-18
 オブジェクト状態管理の委譲 6-19
 クローズ 6-18
XA リソース・マネージャのオープン 6-18
XA リソース・マネージャのクローズ 6-18

あ

アクセサ
 要件の計算 4-65
アプリケーション制御の非活性化
 概要 1-17
 例 3-13
アプリケーションのスケーリング 8-5
 機能のまとめ 8-2

い

- 一時オブジェクト 3-21
- 入れ子になったトランザクション 6-22
- 印刷、製品のマニュアル xiii
- インターフェイス
 - インプリメンテーションのデレゲーション 2-36
 - 検証 2-32
 - コンパイルの制限 2-9
 - 定義 1-3
- インターフェイス・リポジトリ 1-3
- インターフェイス・リポジトリ識別子 1-6
- インプリメンテーション・オブジェクト、オブジェクト・インプリメンテーションを参照 1-2
- インプリメンテーション・コンフィギュレーション・ファイル (ICF ファイル)
 - ICF ファイルを参照 2-9

え

- 永続状態の処理
 - 例 3-14

お

- オブジェクト
 - always トランザクションへの対応 (例) 7-15
 - always トランザクションへの対応 6-14
 - 一時的 3-21
 - インスタンス化 1-8
 - インターフェイスのインプリメント 1-4
 - 活性化 1-25
 - 活性化方針の設定 1-14
 - 管理 1-13
 - サーバントのプール 2-35
 - 状態を持たない 8-29
 - 状態を持つ 8-30

- 状態データの読み取りと書き込み 1-19
- トランザクションから除外 6-16
- トランザクションでのポーリング 6-20
- トランザクションに任意に含める 6-15
- トランザクションの省略 6-17
- トランザクション・バウンド 1-15
- 非活性化 1-25
- プロセスの非活性化 1-17
- プロセス・バウンド 1-15
- メソッド・バウンド 1-15
- レガシー 2-36
- オブジェクト ID
 - OID を参照 3-9
- オブジェクト・インプリメンテーション
 - オブジェクトを参照 1-2
 - 概要 1-2
 - デレゲーション 2-36
- オブジェクトごとのスレッド 4-26, 4-28
- オブジェクト状態
 - BEA Tuxedo システム 1-13
- オブジェクト状態管理
 - Basic サンプルでの管理 3-12
 - XA リソース・マネージャへの委譲 6-19
 - スケラビリティ 8-11
 - トランザクション 6-9
- オブジェクトのインスタンス化 1-8
- オブジェクト・ファクトリ
 - ファクトリを参照
- オブジェクト・リファレンス
 - 関連情報 1-5
 - 作成 2-13
 - 寿命 1-7
 - 生成 1-11
 - 生成 (例) 3-9
 - 内容 1-6
- オブジェクト・リファレンスの作成 2-13
- オブジェクト・リファレンスの生成 1-11

か

カーソル

データベース 6-14

開発プロセス

概要 2-2

会話

トランザクションによるインプリメン
ト 6-2

カスタマ・サポート情報 xiv

活性化されたオブジェクト

最大数の指定 4-66

活性化方針

method 8-11

process 3-12

transaction 6-20

環境変数

検証 4-48

設定 4-47

ディレクトリ・パス 4-47

管理パラメータ

CONCUR_STRATEGY 4-65

MAXACCESSORS 4-64

MAXDISPATCHTHREADS 4-63

MINDISPATCHTHREADS 4-63, 4-65

関連情報 xiii

き

機能

マルチスレッド・サーバ・アプリケー
ション 4-32

く

クライアント・アプリケーション

オブジェクトのアクセス方法 1-5

グループ

サーバのコンフィギュレーション 8-8

作成 8-8

ルーティング要求 8-15

こ

コールバック・メソッド

エラー状態の検出 2-31

コンテキスト・サービス

目的 4-32

コンビニエンス・マクロ 4-53

コンフィギュレーション・ファイル 4-35

制御パラメータ 4-35

設定 4-29

さ

サーバ・アプリケーション

開発 1-11

グループ内での複製 8-5

グループのコンフィギュレーション
8-8

スケーリング 8-5

サーバ・アプリケーションの作成

概要 2-2

サーバ・グループ

コンフィギュレーション 8-8

サーバ・スケルトン

スケルトンを参照 1-3

サーバ・プロセス

複製 8-5

サーバ・プロセスの複製 8-5

サーバント 4-25, 4-39, 4-41

概要 1-8

作成 2-15

プール 2-34

再帰的トランザクション 6-22

再帰的呼び出し 4-25, 4-30

作業ディレクトリ 4-49, 4-53, 4-55, 4-56,
4-62

サポート

技術情報 xiv

し

シグナル 4-40

状態データ

オブジェクトの事前活性化 3-20
読み取りと書き込み 1-19
状態を持つオブジェクト
選択の基準 8-30
定義 1-13
プロセス・バウンド・オブジェクトと
トランザクション・バウン
ド・オブジェクトを参照 1-13
状態を持たないオブジェクト
選択の基準 8-29
定義 1-13
メソッド・バウンド・オブジェクトを
参照 1-13
シングル・スレッド・サーバ
振る舞い 4-35
シングル・スレッド実行可能ファイル
4-37

す

スケルトン
コンパイルの制限 2-9
スレッド
コンテキスト情報 4-32
再帰的呼び出し 4-30
同時要求 4-43
並列処理 4-43
スレッド・セーフ 4-28
スレッドの割り当て 4-29
スレッドの割り当て解除 4-29
スレッド・プール 4-29
最小サイズ 4-29
サイズ 4-29
サイズの超過 4-29
最大サイズ 4-29
最大サイズの設定 4-29
システム・リソースの消費 4-29
スレッドの解放 4-29
スレッドの割り当て 4-29
複数の要求のためのスレッドの再利用
4-30
スレッド・モデル
オブジェクトごとのスレッド、要求ご

とのスレッド 4-28
指定 4-65
要求ごとのスレッド・モデル 4-26,
4-28

せ

セキュリティと CORBA サーバ・アプリ
ケーション 5-2
セキュリティ・モデル
サーバ・アプリケーションでのインプ
リメンテーション 5-3

ち

チューニングとスケーリング 4-67

て

ディレクトリ・パス 4-47
データ依存型ルーティング
ファクトリ・ベース・ルーティングを
参照 8-15
データ破損
危険性 4-39
データベース
オープンとクローズ 2-15
データベース・カーソル 6-14
データ・マーシャリング
無効化 3-20
デザイン・パターン
List-Enumerator 1-30
List-Enumerator (例) 3-18
Process-Entity 1-30
Process-Entity (例) 3-18
University サンプルでの使用 3-17
デバッグのヒント 2-24

と
同時アクセス 4-39
同時要求 4-43
トランザクション

CORBA サーバ・アプリケーションでの
インプリメント 6-5

入れ子になった 6-22

オブジェクト状態の管理 6-19

概要 6-2

会話 6-2

再帰的 6-22

別のスレッドへの受け渡し 4-31

トランザクション状態

スレッドへの関連付け 4-31

トランザクションに關与するオブジェク
ト

定義 6-14

トランザクションの拒否 6-20

トランザクション方針

always 6-14

always (例) 7-15

ICF ファイルでの割り当て 6-18

ignore 6-17

never 6-16

optional 6-15

トランザクション・マネージャ・サーバ

TMS を参照 6-11

ファクトリ・ベース・ルーティング

UBBCONFIG ファイル 8-16

概要 8-14

しくみ 8-15

ファクトリでのインプリメンテーショ
ン 8-19

プール

サーバント 2-34

へ

並行性

手法 4-52, 4-53

メカニズム 4-28

並列処理 4-25, 4-43

ま

マニュアル、入手先 xiii

マルチスレッド・モデル

仕様 4-31

定義 4-31

ゆ

ユーザ定義の例外 6-24

よ

要求ごとのスレッド・サーバ

インプリメンテーション 4-44

要求ごとのスレッド・モデル 4-26, 4-28,
4-65

要求レベルのインターセプタ 4-40

り

リエントラント 4-30

デフォルト設定 4-30

同時実行規則 4-30

リエントラント・サーバント 4-30

オブジェクト状態の保護 4-41

関連情報 4-30

ぬ

ヌル・リソース・マネージャ 6-20

ひ

非リエントラント・サーバント 4-39

ふ

ファクトリ

1-5

概要 1-11

クライアントが取得するしくみ 1-12

登録 2-13

ファクトリ・ベース・ルーティング
8-19

メリット 1-12

例 3-9

作成 4-41
リスナ / ハンドラ
 IIOP 8-2
リソース・マネージャ
 XA のオープン 6-18
 オブジェクト状態管理の委譲 6-19
 ヌル 6-20
 6-18

る

ルーティング
 ファクトリ・ベース、ファクトリ・
 ベース・ルーティングを参照
 8-16
ルーティング基準
 ファクトリでの指定 8-19

れ

例外
 activate_object() 2-27
 ActivateObjectFailed 2-26
 AlreadyRegistered 2-26
 BAD_OPERATION 2-27
 CannotProceed 2-26
 CORBA 2-25
 create_servant 2-27
 CreateServantFailed 2-26
 deactivate_object() 2-27
 DeactivateObjectFailed 2-26
 IllegalInterface 2-26
 InitializeFailed 2-26
 INVALID_TRANSACTION 6-22
 InvalidDomain 2-26
 InvalidInterface 2-26
 InvalidName 2-26
 InvalidObject 2-26
 InvalidObjectID 2-26
 InvalidServant 2-26
 NilObject 2-26
 NoSuchElement 2-26
 OBJ_ADAPTER 6-22

OBJECT_NOT_EXIST 2-27
OrbProblem 2-26
OutOfMemory 2-26
Overflow 2-26
RegistrarNotAvailable 2-26
ReleaseFailed 2-26
TpfProblem 2-26
UnknownInterface 2-26
UserExceptions 2-26
クライアント・アプリケーション
 2-25
サーバ・アプリケーション 2-25
ユーザ定義を記述する方法 6-24
レガシー・オブジェクト
 BEA Tuxedo CORBA への統合 2-36

ろ

ログ・ファイル 4-58

